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 `~/OCRL/HW2_S25`
```

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:

$$x = [p_1, p_2, v_1, v_2] \quad (1)$$
$$u = [a_1, a_2] \quad (2)$$

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

$$\dot{x} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} x + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \quad (3) u$$

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)

    # TODO: discretize this linear system using the Matrix Exponential

    ode_mat = exp([Ac*dt Bc*dt; zeros(nu,nx) zeros(nu,nu)])

    A = ode_mat[1:nx, 1:nx] # TODO
    B = ode_mat[1:nx, nx+1:end] # TODO


    @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  5.2s
Test.DefaultTestSet("discrete time dynamics", Any[], 1, false, false, true, 1.739658545317046e9, 1.739658550542337e9, false, "/home/burger/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:

$$\min_{x_{1:N},u_{1:N-1}} \quad \sum_{i=1}^{N-1} \left[\frac{1}{2} x_i^T Q x_i + \frac{1}{2} u_i^T R u_i \right] + \frac{1}{2} x_N^T Q_f x_N \tag{4}$$

$$\text{st} \quad x_1 = x_{\text{IC}} \tag{5}$$

$$x_{i+1} = A x_i + B u_i \quad \text{for } i = 1, 2, \ldots, N-1 \tag{6}$$

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
```

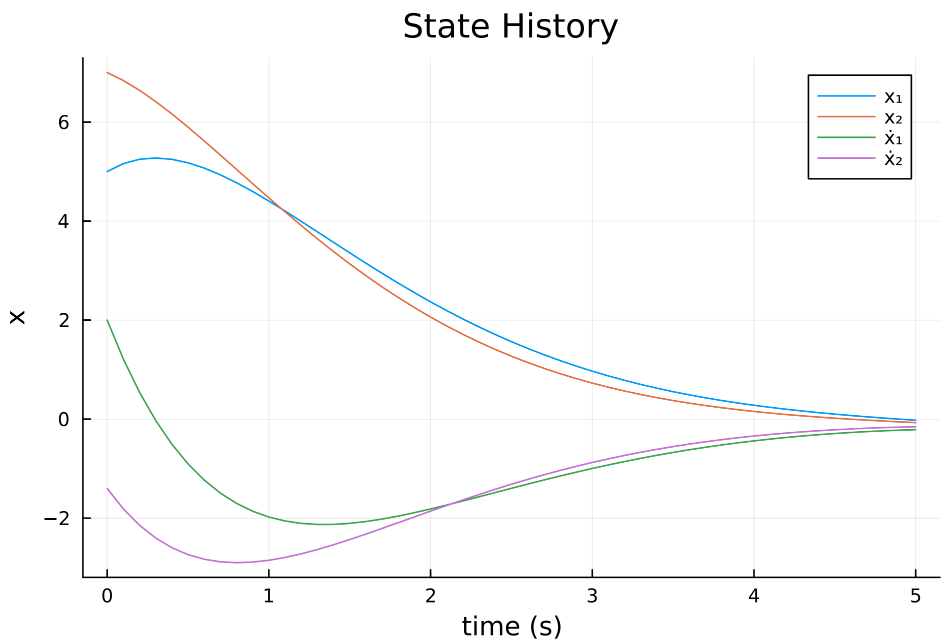
```julia
"""
X,U = convex_trajopt(A,B,Q,R,Qf,N,x_ic; verbose = false)

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)
        x_k = X[:,k]
        u_k = U[:,k]

        quadform_x = 0.5 * cvx.quadform(x_k,Q)
        quadform_u = 0.5 * cvx.quadform(u_k,R)

        # add stagewise cost
        cost += quadform_x + quadform_u
    end

    # add terminal cost
    cost += 0.5 * 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 = vcat(prob.constraints, (X[:,1] == x_ic))
    for k = 1:(N-1)
        # dynamics constraints
        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)
```

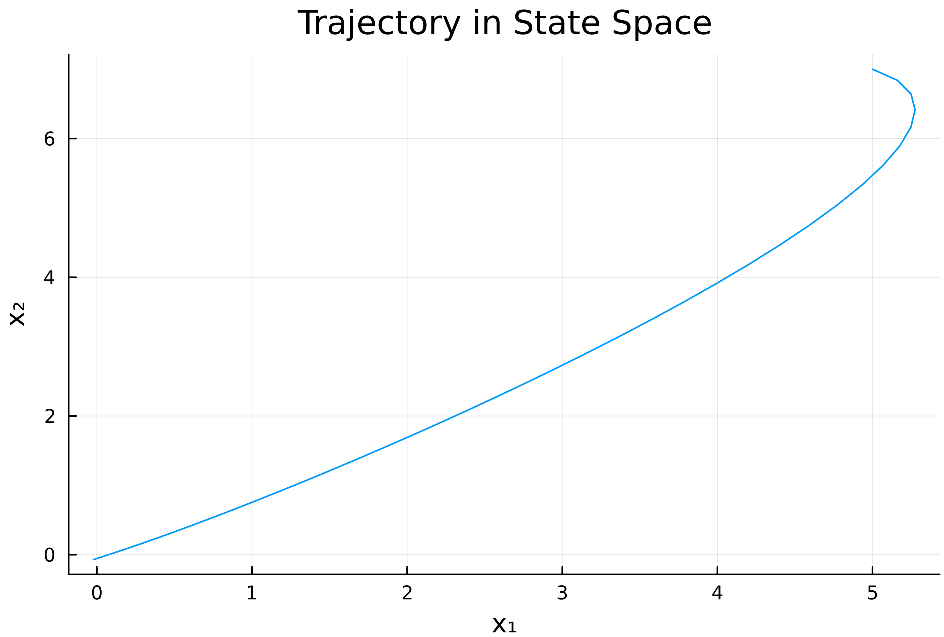
```julia
    return X, U
end
```

```
convex_trajopt
```

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

In [5]:
```julia
@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

    # TODO dynamics simulation
    for k = 1:(N-1)
        Xsim[k+1] = A*Xsim[k] + B*Ucvx[k]
    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  15.3s
Test.DefaultTestSet("LQR via Convex.jl", Any[], 6, false, false, true, 1.73965855117871e9, 1.73965856652
8077e9, false, "/home/burger/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:

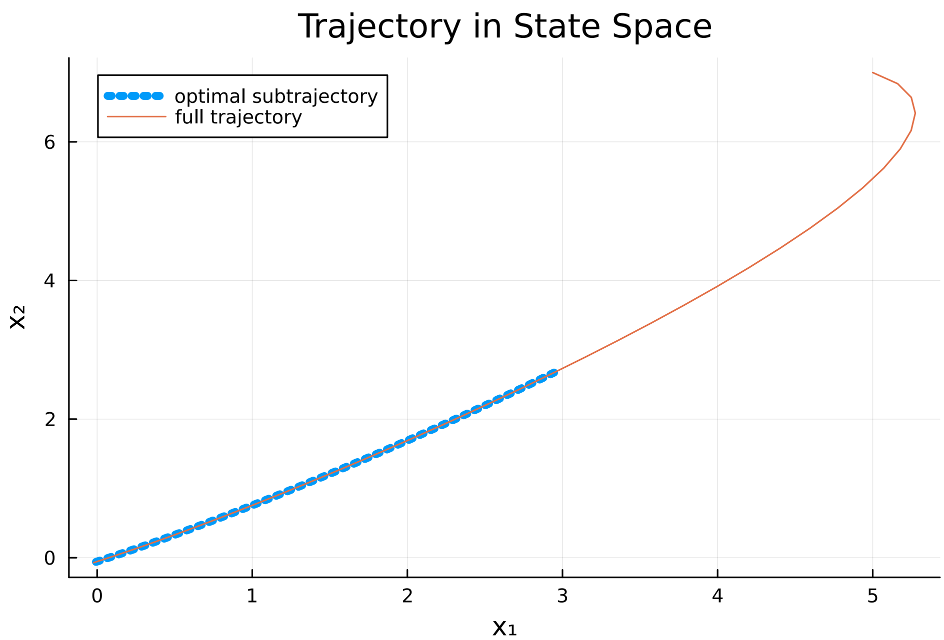
$$\begin{aligned} \min_{x_{1:N},u_{1:N-1}} \quad & \sum_{i=1}^{N-1} \left[\frac{1}{2} x_i^T Q x_i + \frac{1}{2} u_i^T R u_i \right] + \frac{1}{2} x_N^T Q_f x_N && (7) \\ \text{st} \quad & x_1 = x_{\text{IC}} && (8) \\ & x_{i+1} = A x_i + B u_i \quad \text{for } i = 1, 2, \ldots, N-1 && (9) \end{aligned}$$

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 $x_{L:N}, u_{L:N-1}$:

$$\begin{aligned}
\min_{x_{L:N},u_{L:N-1}} \quad & \sum_{i=L}^{N-1} \left[\frac{1}{2} x_i^T Q x_i + \frac{1}{2} u_i^T R u_i \right] + \frac{1}{2} x_N^T Q_f x_N && (10)\\
\text{st} \quad & x_L = x_L^* && (11)\\
& x_{i+1} = A x_i + B u_i \quad \text{for } i = L, L+1, \ldots, N-1 && (12)
\end{aligned}$$

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  0.4s
Test.DefaultTestSet("Bellman's Principle of Optimality", Any[], 4, false, false, true, 1.739658566542699
e9, 1.739658566909096e9, false, "/home/burger/OCRL/HW2_S25/jl_notebook_cell_df34fa98e69747e1a8f8a730347b
8e2f_X16sZmlsZQ==.jl")
```

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

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

$$\min_{x_{1:N},u_{1:N-1}} \quad \sum_{i=1}^{N-1} \left[\frac{1}{2}x_i^TQx_i + \frac{1}{2}u_i^TRu_i\right] + \frac{1}{2}x_N^TQ_fx_N \tag{13}$$

$$\text{st} \quad x_1 = x_{\text{IC}} \tag{14}$$

$$x_{i+1} = Ax_i + Bu_i \quad \text{for } i = 1,2,\ldots,N-1 \tag{15}$$

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

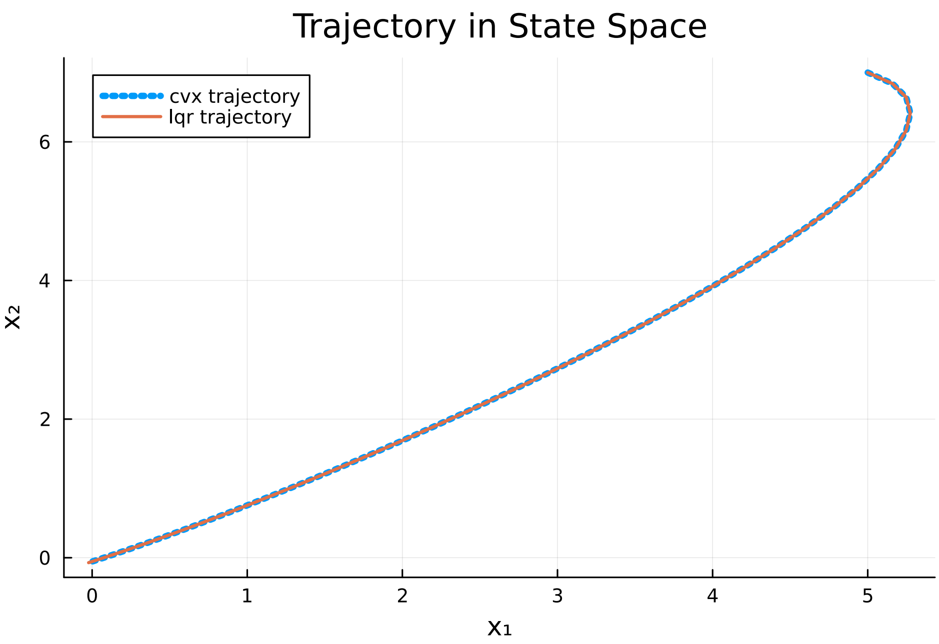
$V_k(x) = \frac{1}{2}x^TP_kx.$

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
        # TODO
        K[k] = inv(R + B'*P[k+1]*B)*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
```

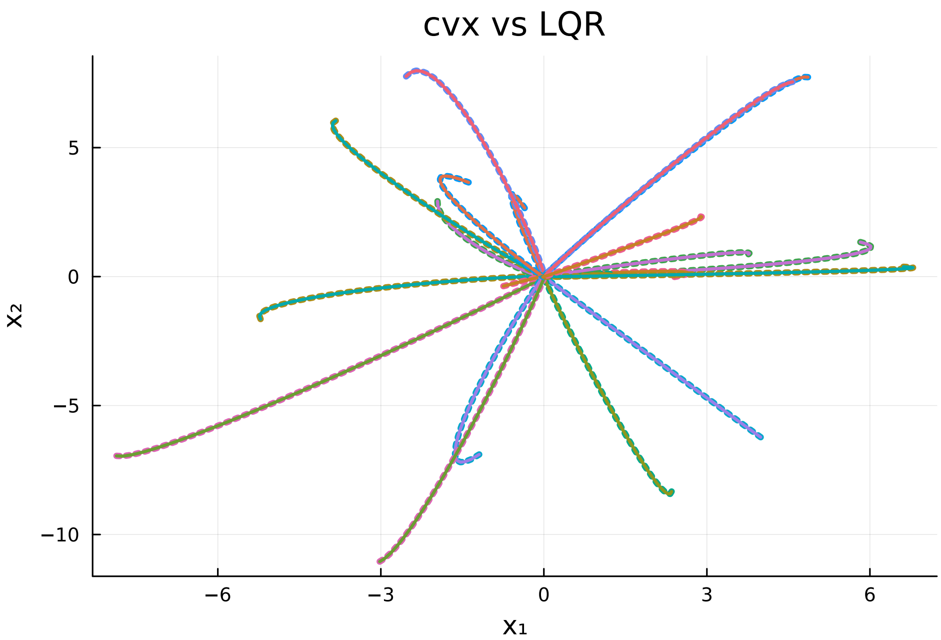



```
Test Summary:          | Pass  Total  Time
Convex trajopt vs LQR |    3      3  1.0s
Test.DefaultTestSet("Convex trajopt vs LQR", Any[], 3, false, false, true, 1.739658566927528e9, 1.739658
567916644e9, false, "/home/burger/OCRL/HW2_S25/jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X22sZml
sZQ==.jl")
```

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]:
```julia
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  0.5s
Test.DefaultTestSet("Convex trajopt vs LQR", Any[], 40, false, false, true, 1.739658567970644e9, 1.739658568478028e9, false, "/home/burger/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 $u = -K(x - x_{goal})$

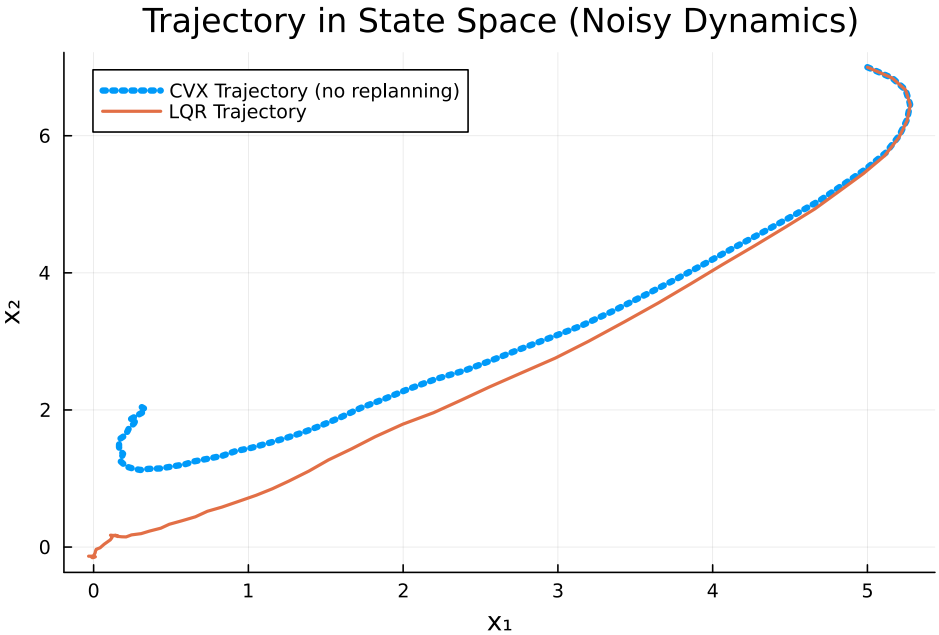
First we are going to look at a simulation with the following white noise:

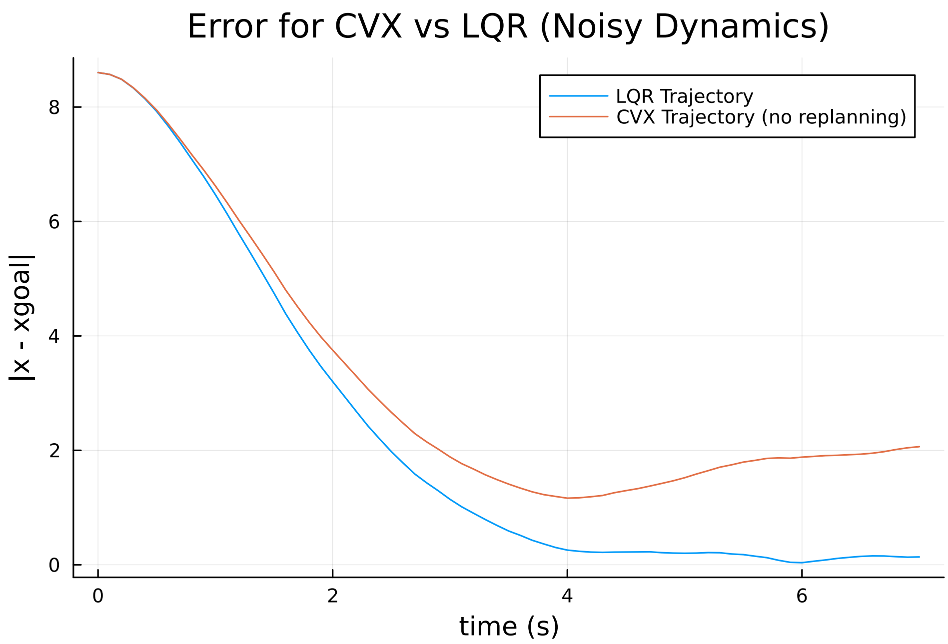
$$x_{k+1} = Ax_k + Bu_k + \text{noise}$$

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

In [10]:
```
@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
```

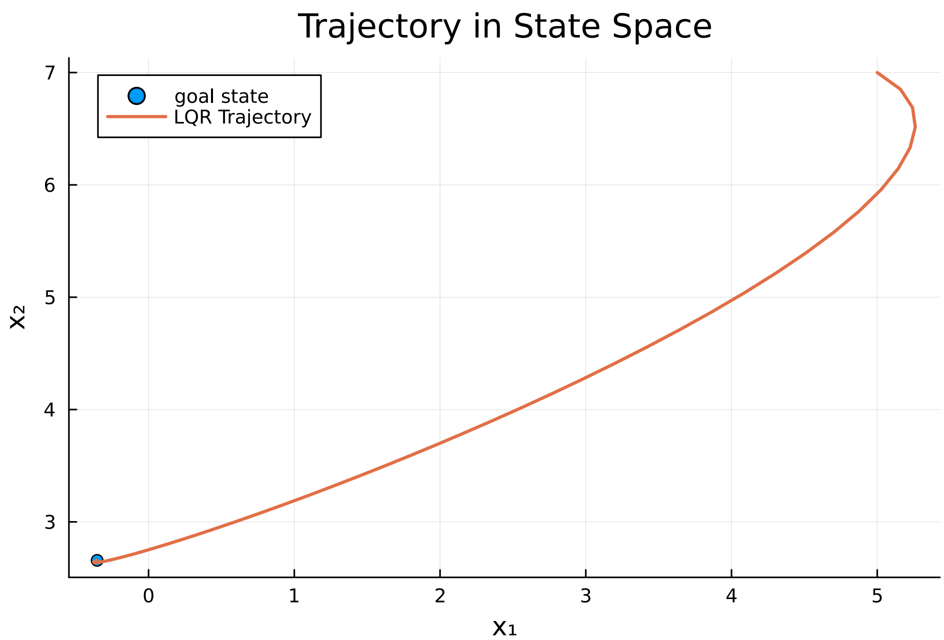
```
    Xsim_lqr = [zeros(nx) for i = 1:N]
    Xsim_lqr[1] = 1*x0
    for i = 1:N-1
        # sampled noise to be added after each step
        noise = [.005*randn(2);.1*randn(2)]

        # 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  0.5s
Test.DefaultTestSet("Why LQR is great reason 1", Any[], 3, false, false, true, 1.739658568495777e9, 1.73
9658568987856e9, false, "/home/burger/OCRL/HW2_S25/jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X26
sZmlsZQ==.jl")
```

In [15]:
```
@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 = [5*randn(2);0;0]
    @show xgoal
    @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
```




```
xgoal = [-0.35291569476948953, 2.6573837689159814, 0.0, 0.0]
Test Summary:              | Pass  Total  Time
Why LQR is great reason 2 |    3      3  0.1s
Test.DefaultTestSet("Why LQR is great reason 2", Any[], 3, false, false, true, 1.73965886652272e9, 1.739658866585057e9, false, "/home/burger/OCRL/HW2_S25/jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X30sZmlsZQ==.jl")
```

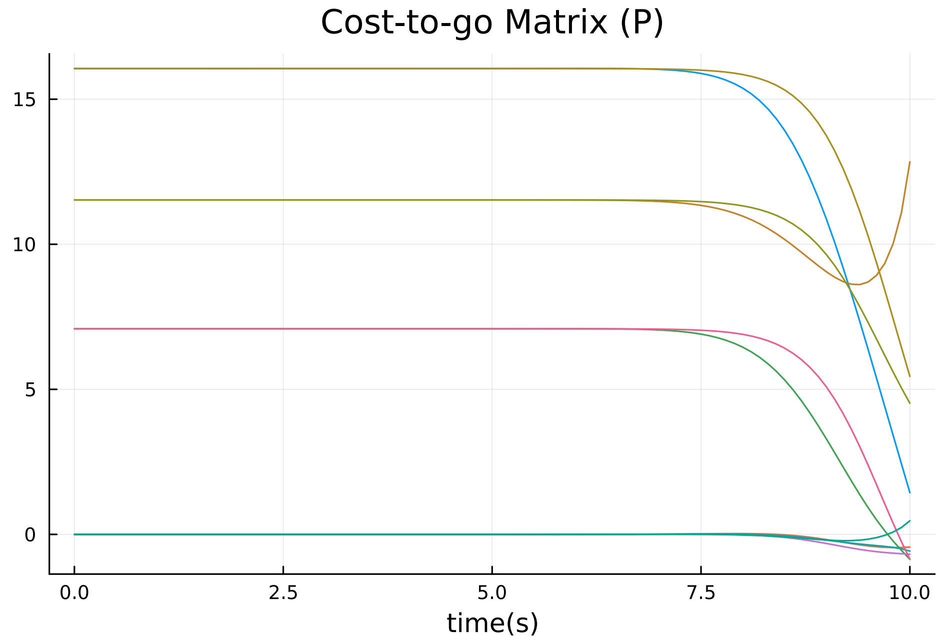
Part E: Infinite-horizon LQR (10 pts)

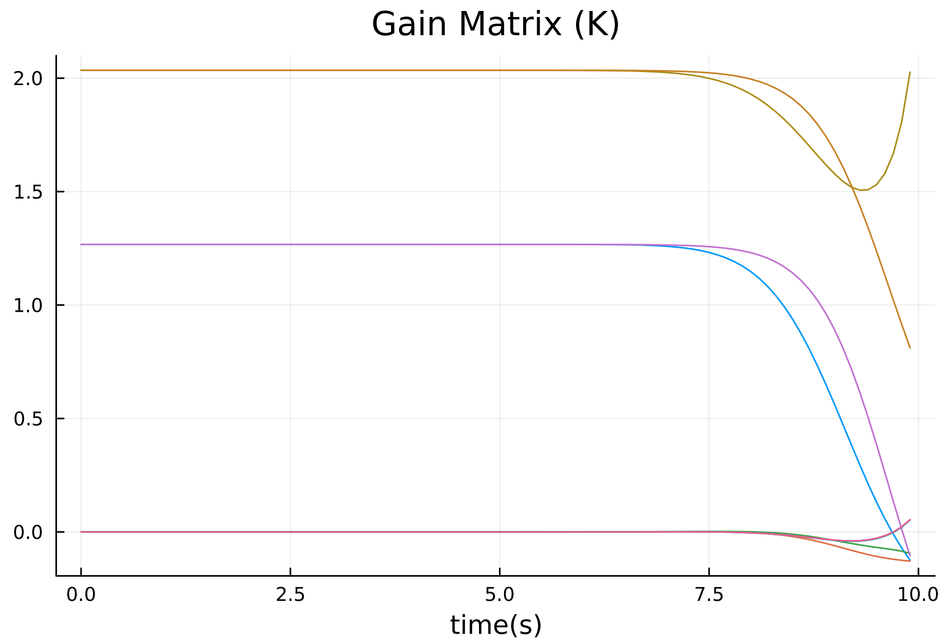
Up until this point, we have looked at finite-horizon LQR which only considers a finite number of timesteps in our trajectory. When this problem is solved with a Riccati recursion, there is a new feedback gain matrix K_k for each timestep. As the length of the trajectory increases, the first feedback 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 [16]:
```julia
# 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  0.1s
Test.DefaultTestSet("P and K time analysis", Any[], 2, false, false, true, 1.739658989691477e9, 1.739658
98976154e9, false, "/home/burger/OCRL/HW2_S25/jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X32sZmls
ZQ==.jl")
```

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

$$\|P_k - P_{k+1}\| \leq \text{tol}$$

And return the steady state P and K.

In [18]:
```
"""
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)

    # Riccati
    for riccati_iter = 1:max_iter

        # TODO
        K = inv(R + B'*P*B)*B'*P*A
        P_new = Q + A'*P*(A - B*K)
        # check for convergence
        if norm(P - P_new) < tol
            return P, 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  0.7s
Test.DefaultTestSet("ihlqr test", Any[], 3, false, false, true, 1.739659157979138e9, 1.739659158683096e9
, false, "/home/burger/OCRL/HW2_S25/jl_notebook_cell_df34fa98e69747e1a8f8a730347b8e2f_X34sZmlsZQ==.jl")
```