```
In [1]: from IPython import display
        import matplotlib.pyplot as plt
        from matplotlib.lines import Line2D
        from gym.envs.mujoco import *
        from envs.hopper_env import HopperModEnv
        from envs.cheetah_env import CheetahModEnv
        import numpy as np
        import copy
        import gym
        from scipy.io import loadmat
        from scipy.io import savemat
        import moviepy.editor as mpy
        from simulators import *
        from rot_utils import *
        import seaborn as sns
        sns.set_style('darkgrid')
        import warnings
        warnings.filterwarnings('ignore')
```

(a) LQR for Linear Systems

Let's start with a linear system:

```
In [2]: A = np.array([[0.0481, -0.5049, 0.0299, 2.6544, 1.0608],
                      [2.3846, -0.2312, -0.1260, -0.7945, 0.5279],
                      [1.4019, -0.6394, -0.1401, 0.5484, 0.1624],
                      [-0.0254, 0.4595, -0.0862, 2.1750, 1.1012],
                      [0.5172, 0.5060, 1.6579, -0.9407, -1.4441]])
        B = np.array([[-0.7789, -1.2076],
                      [0.4299, -1.6041],
                      [0.2006, -1.7395],
                      [0.8302, 0.2295],
                      [-1.8465, 1.2780]])
        dx = A.shape[0]
        du = B.shape[1]
```

Now verify the system is controllable.

To do so we need to check if rank($[B\ AB\ A^2B\ A^3B\ A^4B]$) == 5. (generally we'd go up to $A^{(n-1)}B$, and verify rank == n)

```
In [3]: # verify the above statement
        lst = [B]
        """YOUR CODE HERE"""
        An = np.identity(dx)
        for i in range(1, dx):
            An = An.dot(A)
            AnB = An.dot(B)
            lst.append(AnB)

        """YOUR CODE ENDS HERE"""
        np.linalg.matrix_rank(np.hstack(lst))

Out[3]: 5
```

Recall the following optimal control problem:

$$\min_{x,u} \sum_{t=0}^{T-1} (x_t' Q x_t + u_t' R u_t) + x_T' Q_{final} x_T$$
$$\text{s.t. } x_{t+1} = A x_t + B u_t$$

For T going to infinity, we'll have that the Value Function and the Feedback Controller at time $t = 0$ reach a steady-state --- the optimal value function and feedback controller for infinitely many time-steps to-go

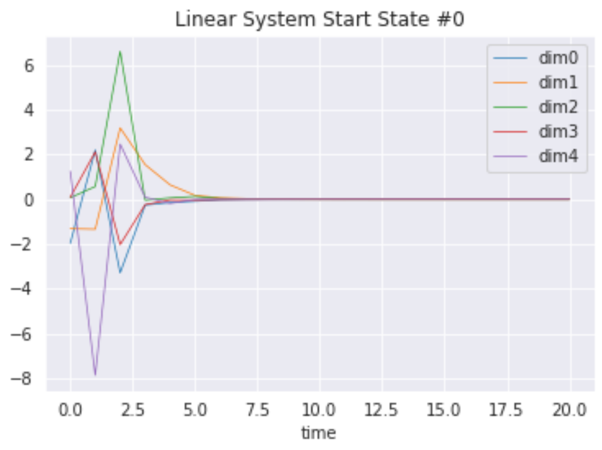
So let's run the Value Iteration Solution to the LQR control problem until it has converged, and then use that infinite horizon optimal feedback controller to stabilize our system at 0.

In [4]:
```
# implement the infinite horizon optimal feedback controller
def lqr_infinite_horizon(A, B, Q, R):
    """
    find the infinite horizon K and P through running LQR back-ups
    until l2-norm(K_new - K_curr, 2) <= 1e-4
    return: K, P
    """

    dx, du = A.shape[0], B.shape[1]
    P, K_current = np.eye(dx), np.zeros((du, dx))

    """YOUR CODE HERE"""
    while True:
        K_new = - np.linalg.inv(R + B.T @ P @ B) @ B.T @ P @ A
        P = Q + K_new.T @ R @ K_new + (A + B @ K_new).T @ P @ (A + B @ K_new)
        if np.linalg.norm(K_new - K_current, 2) <= 1e-4:
            break
        else:
            K_current = K_new

    """YOUR CODE ENDS HERE"""
    return K_new, P
```

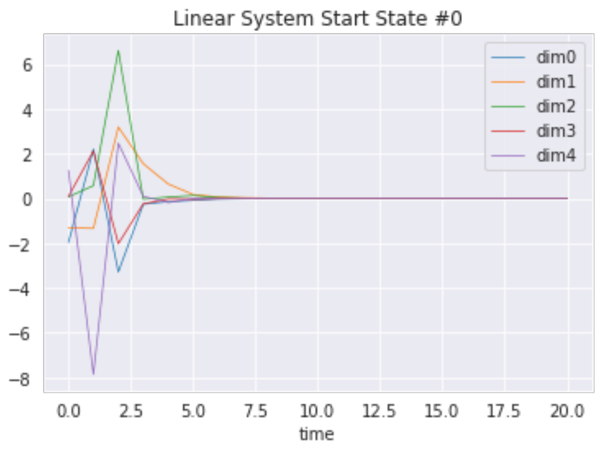
In [5]:
```
# problem has been defined, let's solve it:
Q, R = np.eye(dx), np.eye(du)
K_inf, P_inf = lqr_infinite_horizon(A, B, Q, R)
```

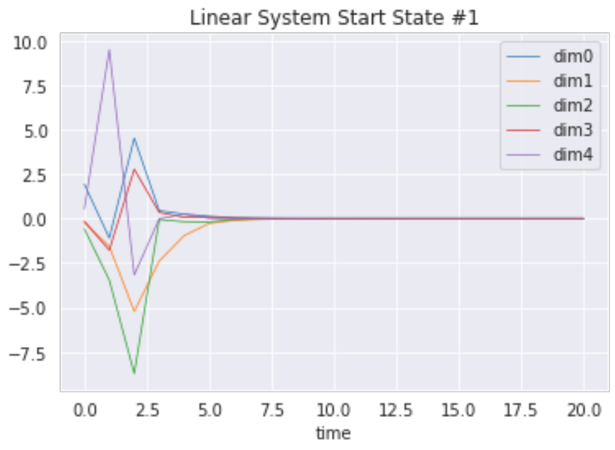
Now let's simulate and see what happens for a few different starting states.

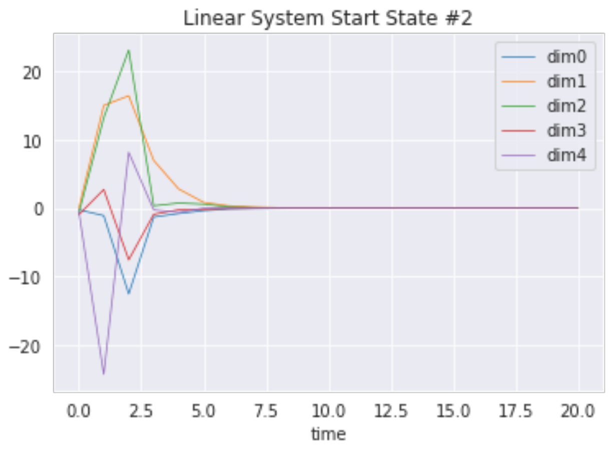
Here's what a reference plot looks like for the *first starting state with no noise* in state dynamics so you may compare:

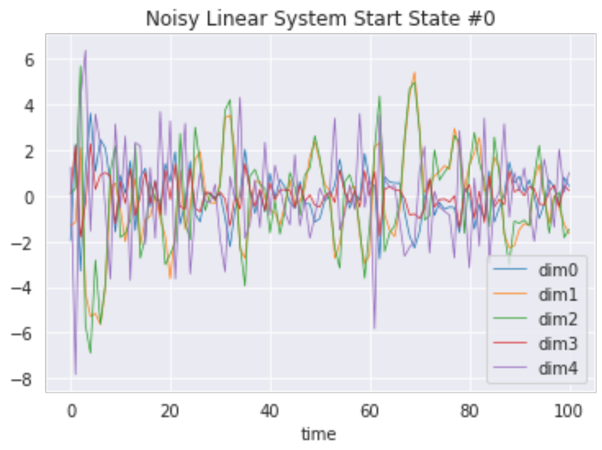


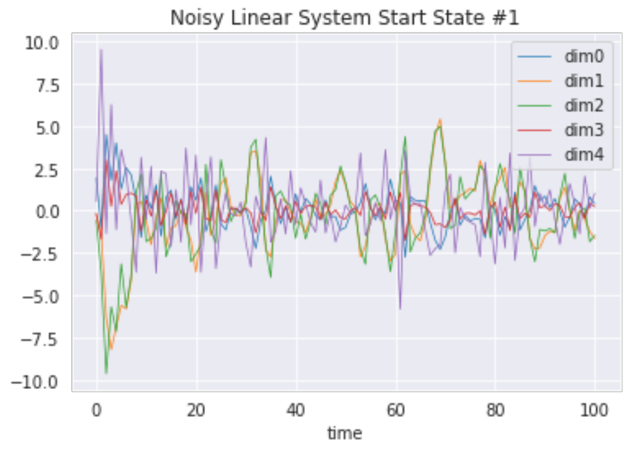
In [6]:
```
# fill in the simulation to use your controller, K_inf, at each timestep then r
un the cell to generate plots
def simulate(A, B, K_inf, n_starting_states, T, noise=None):
    for s in np.arange(n_starting_states):
        x, u = np.zeros((K_inf.shape[1], T+1)), np.zeros((K_inf.shape[0], T+1))
        x[:,0] = starting_states[:,s]
        for t in np.arange(T):
            """YOUR CODE HERE"""
            u[:,t] = K_inf @ x[:,t]
            """YOUR CODE ENDS HERE"""
            x[:,t+1] = A @ x[:,t] + B @ u[:,t]
            if noise is not None:
                x[:,t+1] += noise[:,t]
        plt.plot(x.T, linewidth=.7)
        plt.xlabel('time')
        plt.title("Noisy Linear System Start State #{}".format(s)) if noise is
not None else plt.title("Linear System Start State #{}".format(s))
        plt.legend(["dim"+str(i) for i in range(len(x))])
        plt.show()


starting_states = np.array([[-1.9613, 1.9277, -0.2442],
                            [-1.3127, -0.2406, -0.0260],
                            [0.0698, -0.5860, -0.7522],
                            [0.0935, -0.1524, -0.9680],
                            [1.2494, 0.5397, -0.5146]])
n_starting_states = starting_states.shape[1]
T = 20 # simulating for 20 steps
simulate(A, B, K_inf, n_starting_states, T)

# and in the presence of noise:
noise_id = "p_a_w"
T = 100 # simulating for 100 steps
simulate(A, B, K_inf, n_starting_states, T, noise=loadmat("mats/"+noise_id+".ma
t")[noise_id])
```

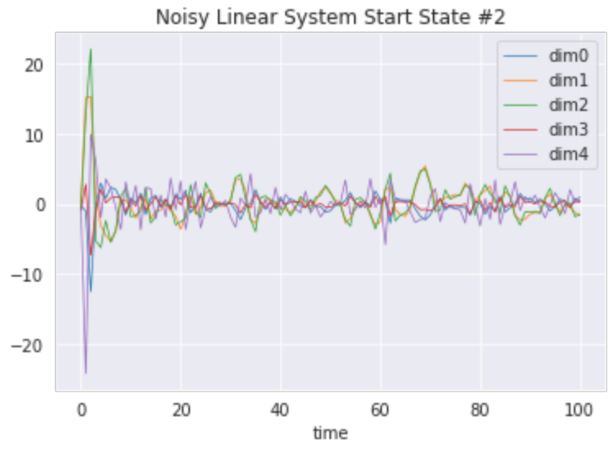

Linear System Start State #0
dim0
dim1
dim2
dim3
dim4
time


Linear System Start State #1
dim0
dim1
dim2
dim3
dim4
time


Linear System Start State #2
dim0
dim1
dim2
dim3
dim4
time







(b) LQR-based Stabilization for Nonlinear Systems

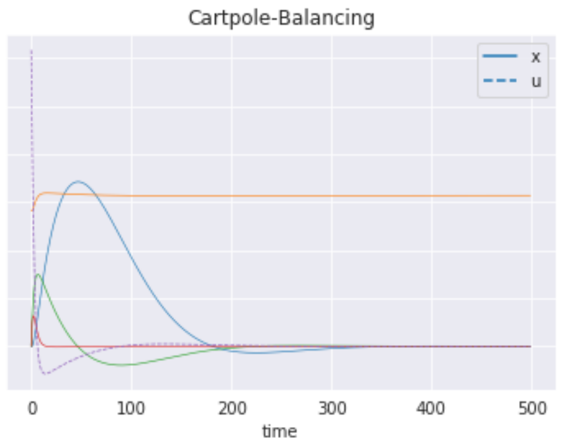
Now let's consider nonlinear systems. Linearize around one point and design an infinite horizon controller for the resulting system.

In [7]:
```
# implement linearization about a point
def linearize_dynamics(f, x_ref, u_ref, dt, my_eps, x_ref_tplus1=None):
    """
    f : dynamics simulator
    my_eps : delta for forward and backward differences you'll need
    NOTE: please use centered finite differences!

    x(:,t+1) - x_ref  approximately = A*( x(:,t)-x_ref ) + B* ( u(:,t) - u_ref
) + c
    If we pick x_ref and u_ref to constitute a fixed point, then c == 0

    For part (b), you do not need to use the optional argument (nor c).
    For part (d), you'll have to revisit and modify this function
        --at this point, you'll want to use the optional argument and the resul
ting c.

    return: A, B, c
    """

    if x_ref_tplus1 is not None:
        x_ref_next = x_ref_tplus1
    else:
        x_ref_next = x_ref

    dx, du = x_ref.shape[0], u_ref.shape[0]
    A, B = np.zeros((dx, dx)), np.zeros((dx, du))

    """YOUR CODE HERE"""
    for j in range(dx):
        xj_forward = x_ref.copy()
        xj_forward[j] = xj_forward[j] + my_eps / 2
        xj_backward = x_ref.copy()
        xj_backward[j] = xj_backward[j] - my_eps / 2
        A[:, j] = (f(xj_forward, u_ref, dt) - f(xj_backward, u_ref, dt)) / (my_
eps)

    for j in range(du):
        uj_forward = u_ref.copy().astype(float)
        uj_forward[j] = uj_forward[j] + my_eps / 2
        uj_backward = u_ref.copy().astype(float)
        uj_backward[j] = uj_backward[j] - my_eps / 2
#         B[:, j] = (f(x_ref, uj_forward, dt) - f(x_ref, uj_backward, dt)) / (m
y_eps)
        B[:, j] = (f(x_ref_next, uj_forward, dt) - f(x_ref_next, uj_backward, d
t)) / (my_eps)
    """YOUR CODE ENDS HERE"""

    c = f(x_ref, u_ref, dt) - x_ref_next
    if len(B.shape) == 1:
        return A, B.reshape(-1, 1), c
    return A, B, c
```

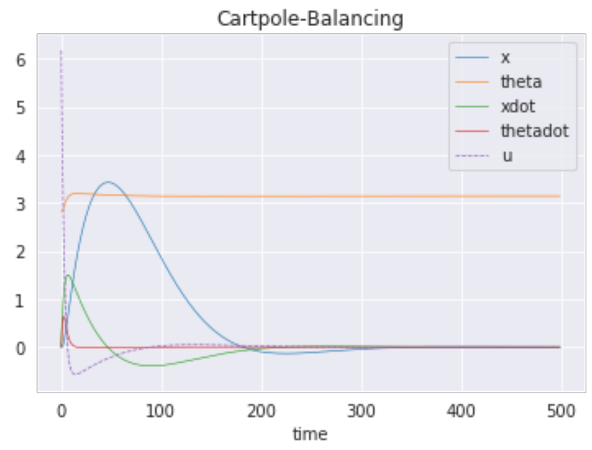
In [8]:
```
# take an environment and find the infinite horizon controller for the lineariz
ed system
def lqr_nonlinear(config):
    env = config['env']
    f = config['f']
    dt = 0.1 # we work with discrete time
    my_eps = 0.01 # finite difference for numerical differentiation

    # load in our reference points
    x_ref, u_ref = config['x_ref'], config['u_ref']

    # linearize
    A, B, c = linearize_dynamics(f, x_ref, u_ref, dt, my_eps)
    dx, du = A.shape[0], B.shape[1]
    Q, R = np.eye(dx), np.eye(du)*2

    # solve for the linearized system
    K_inf, P_inf = lqr_infinite_horizon(A, B, Q, R) # you implemented in part
(a)

    # recognize the simulation code from part (a)? modify it to use your contro
ller at each timestep
    def simulate(K_inf, f, x_ref, u_ref, dt, n_starting_states, T, noise=None):
        for s in np.arange(n_starting_states):
            x, u = np.zeros((K_inf.shape[1], T+1)), np.zeros((K_inf.shape[0],
T+1))
            x[:,0] = starting_states[:,s]
            for t in np.arange(T):
                """YOUR CODE HERE"""
                u[:,t] = u_ref + K_inf @ (x[:, t] - x_ref)
                """YOUR CODE ENDS HERE"""
                x[:,t+1] = f(x[:,t], u[:,t], dt)
                if "p_val" in config.keys():
                    perturbation_values = config["p_val"]
                    perturb = perturbation_values[t//(T//len(perturbation_value
s))]
                    x[:,t+1] = f(x[:,t], u[:,t], dt, rollout=True,perturb=pertu
rb)
                if env is not None:
                    if t % 5 == 0:
                        plt.clf()
                        plt.axis('off')
                        plt.grid(b=None)
                        plt.imshow(env.render(mode='rgb_array', width=256, heig
ht=256))
                        plt.title("Perturbation Magnitude {}".format(perturb))
                        display.clear_output(wait=True)
                        display.display(plt.gcf())

                if noise is not None:
                    x[:,t+1] += noise[:,t]
            if env is not None:
                plt.clf()

            plt.plot(x.T[:-1], linewidth=.6)
            plt.plot(np.squeeze(u.T[:-1])/10.0, linewidth=.7, linestyle='--') #
scaling for clarity
            if 'legend' in config.keys():
                config['legend'].append('u')
                plt.legend(config['legend'])
            else:
                legend_elements = [Line2D([0], [0], label='x'),Line2D([0], [0],
linestyle='--', label='u')]
```

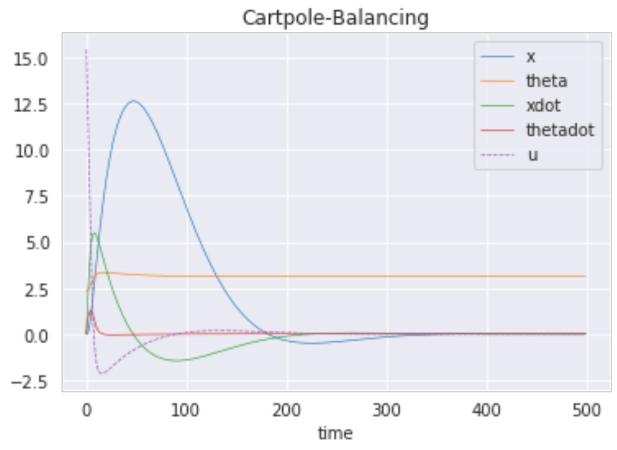
Cartpole-Balancing

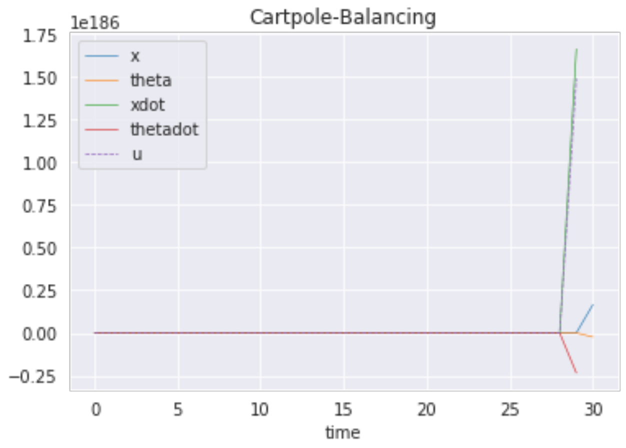
Here's what a reference plot looks like for the *first starting state with no noise* in state dynamics so you may compare:

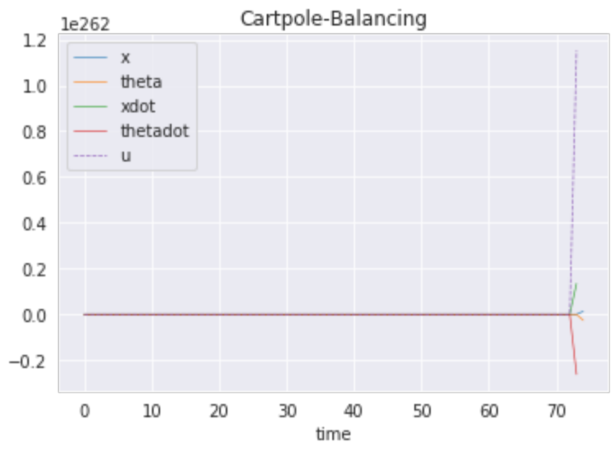


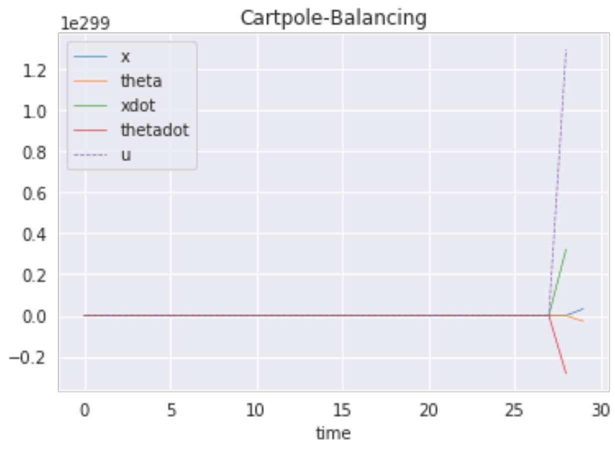
In [9]:
```
# Find the infinite horizon controller for the linearized version of the cartpo
le balancing problem
cartpole_config = {
    'f': sim_cartpole,
    'exp_name': "Cartpole-Balancing",
    'env': None,
    'steps': 500,
    'x_ref': np.array([0, np.pi, 0, 0]),
    'u_ref': np.array([0]),
    'legend':['x', 'theta', 'xdot', 'thetadot'],
    'ss': np.array([[0, 0, 0, 10, 50],
                    [9*np.pi/10, 3*np.pi/4, np.pi/2, 0, 0],
                    [0, 0, 0, 0, 0],
                    [0, 0, 0, 0, 0]]), #ss = starting states
    'noise': 'p_b_w',
}
lqr_nonlinear(cartpole_config)
```

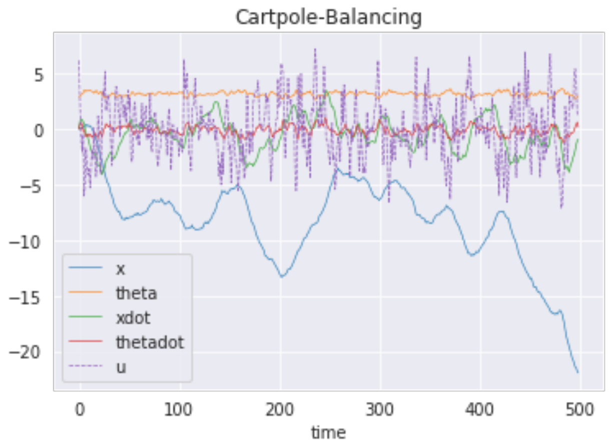

Cartpole-Balancing
x
theta
xdot
thetadot
u
time

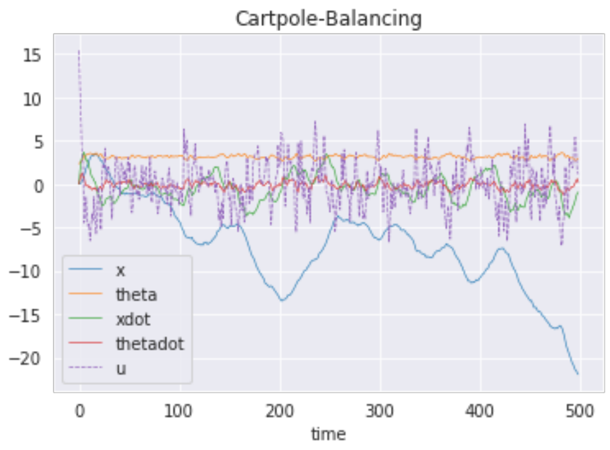

Cartpole-Balancing
x
theta
xdot
thetadot
u
time

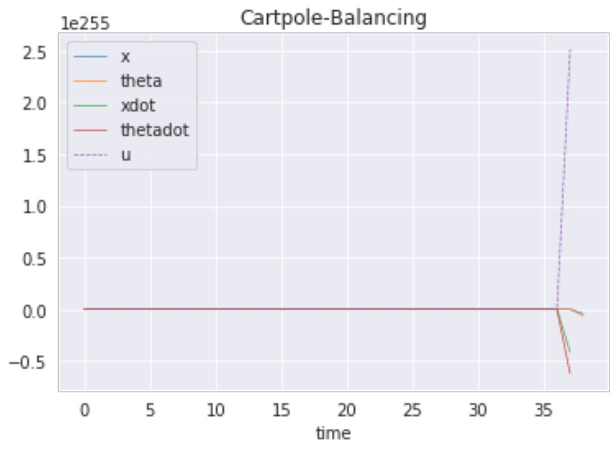

Cartpole-Balancing
1e186
x
theta
xdot
thetadot
u
time

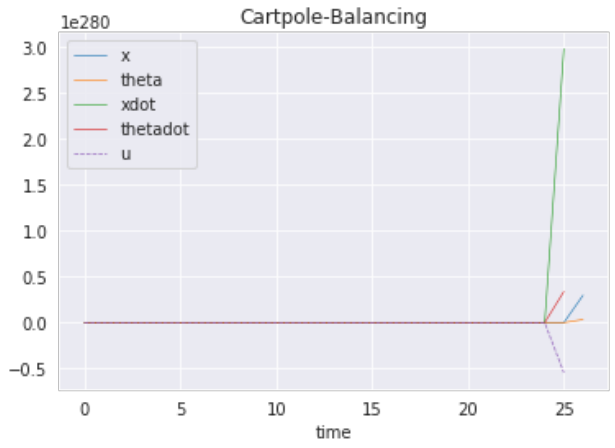

Cartpole-Balancing
1e262
x
theta
xdot
thetadot
u
time

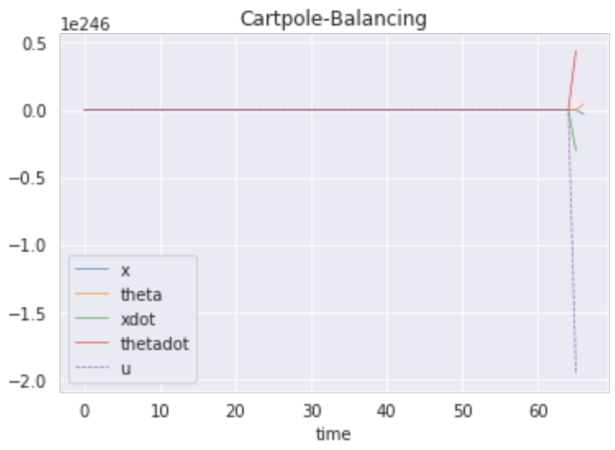

Cartpole-Balancing
1e299
x
theta
xdot
thetadot
u
time


Cartpole-Balancing
x
theta
xdot
thetadot
u
time


Cartpole-Balancing
x
theta
xdot
thetadot
u
time


Cartpole-Balancing
1e255
x
theta
xdot
thetadot
u
time


Cartpole-Balancing
1e280
x
theta
xdot
thetadot
u
time

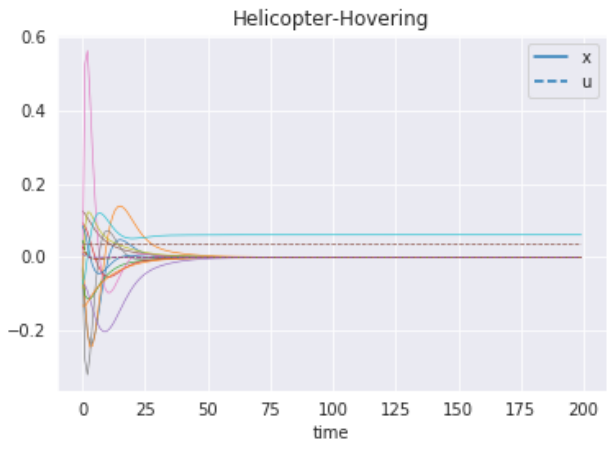


Question: Inspecting the generated plots for the system without noise, explain in 2-3 sentences how the different starting states affect the problem differently. In your answer, include an analysis of one starting state which causes failure and explain why this might happen.

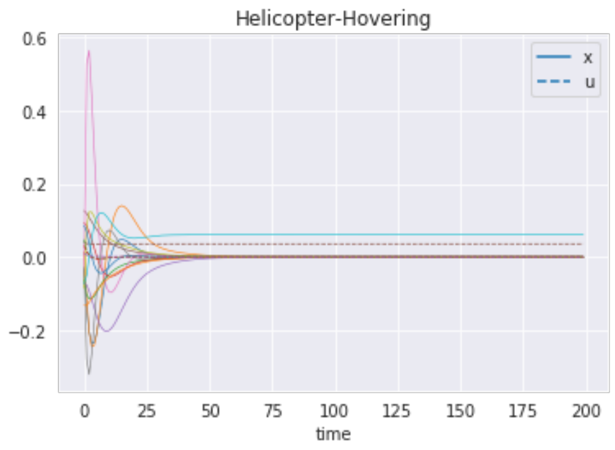
Response: The starting states may be close to or far from the reference state. In the latter case, the linearization using Taylor's expansion drives the system far away from its original nonlinearity, leading to the inaccurate approximation. For instance, the last starting state is [50, 0, 0, 0] which is very far from the reference state [0, pi, 0, 0], resulting in the diverging result.

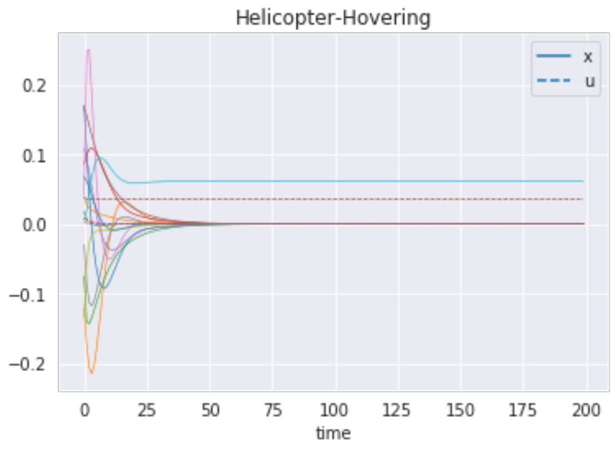
Helicopter in Hover

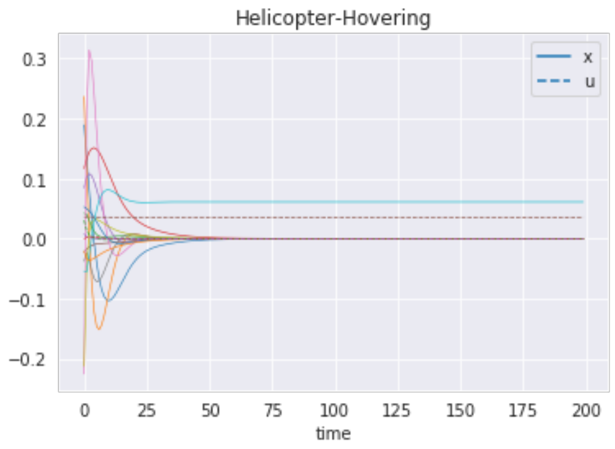
Now let's stabilize a helicopter in hover. We can use the *same* code written for the cartpole system to do so; simply run the cell below which defines the fixed point. Here's what a reference plot looks like for the *first starting state with no noise* in state dynamics so you may compare:

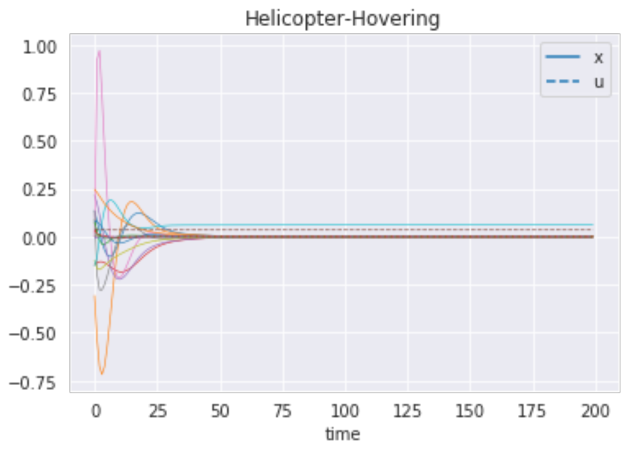


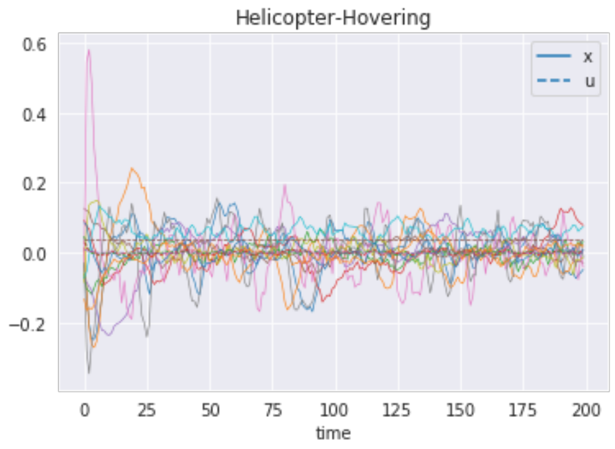
In [10]:
```
# Find the infinite horizon controller for the linearized version of the hoveri
ng copter
# Just run the cell below to generate plots using the code you wrote for cartpo
le!
x_ref, u_ref = np.zeros(12), np.zeros(4)
x_ref[9] = np.arcsin(3.0/(5*9.81))
u_ref[3] = 9.81*5*np.cos(x_ref[9])/137.5
heli_config = {
    'f': sim_heli,
    'env': None,
    'exp_name': "Helicopter-Hovering",
    'steps': 200,
    'x_ref': x_ref,
    'u_ref': u_ref,
    'ss': loadmat("mats/p_c_heli_starting_states.mat")["heli_starting_states"],
#ss = starting states
    'noise': 'p_c_w',
}
lqr_nonlinear(heli_config)
```

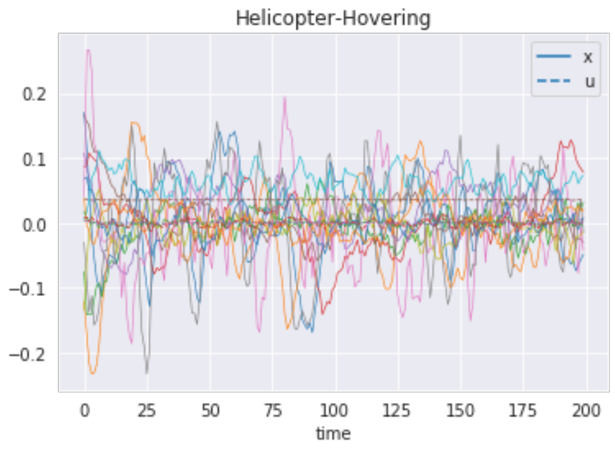

Helicopter-Hovering
x
u
0.6
0.4
0.2
0.0
−0.2
0
25
50
75
100
125
150
175
200
time

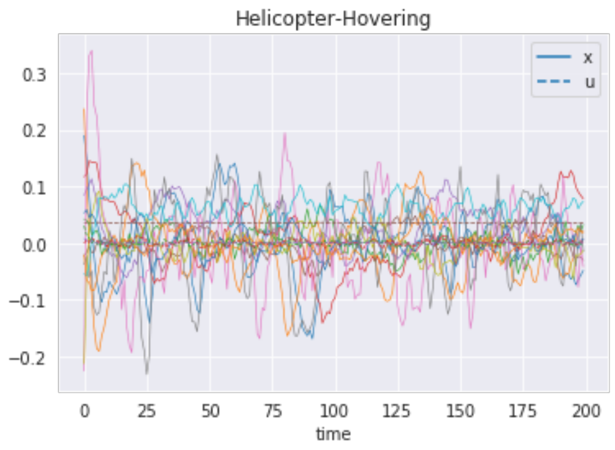

Helicopter-Hovering
x
u
0.2
0.1
0.0
−0.1
−0.2
0
25
50
75
100
125
150
175
200
time

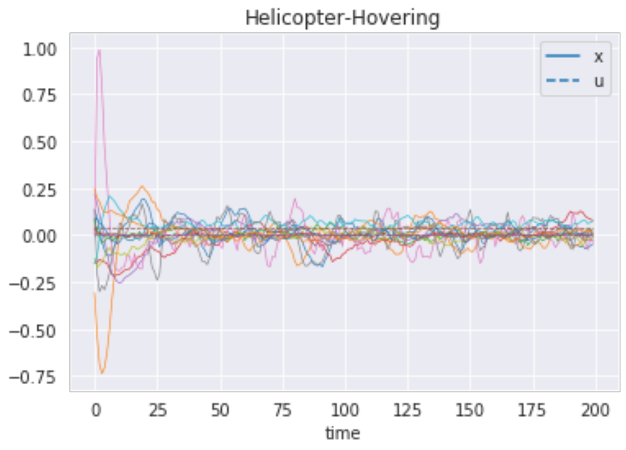

Helicopter-Hovering
x
u
0.3
0.2
0.1
0.0
−0.1
−0.2
0
25
50
75
100
125
150
175
200
time


Helicopter-Hovering
x
u
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
0
25
50
75
100
125
150
175
200
time


Helicopter-Hovering
x
u
0.6
0.4
0.2
0.0
−0.2
0
25
50
75
100
125
150
175
200
time


Helicopter-Hovering
x
u
0.2
0.1
0.0
−0.1
−0.2
0
25
50
75
100
125
150
175
200
time

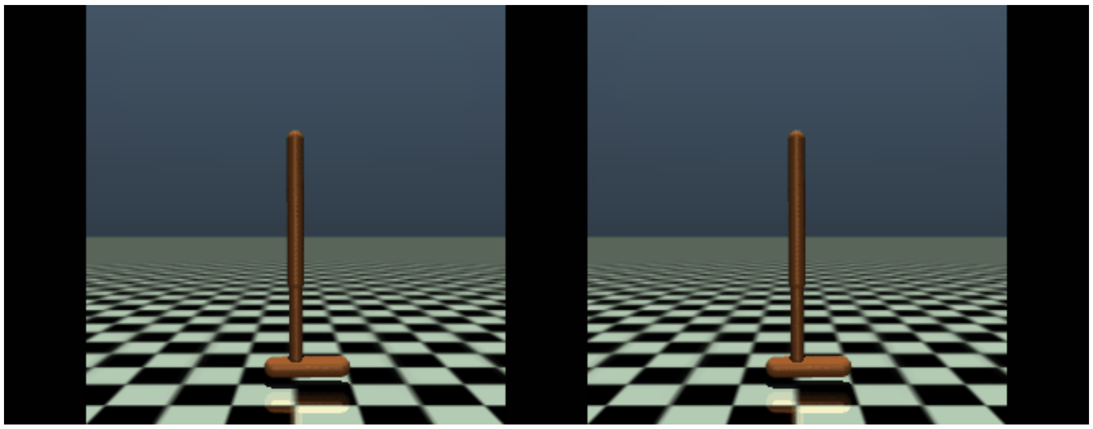




Gym Environments

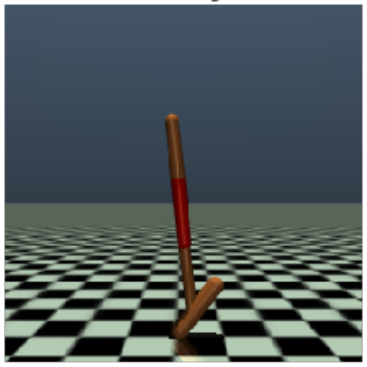
Now we'll test on some environments you may be familiar with now.

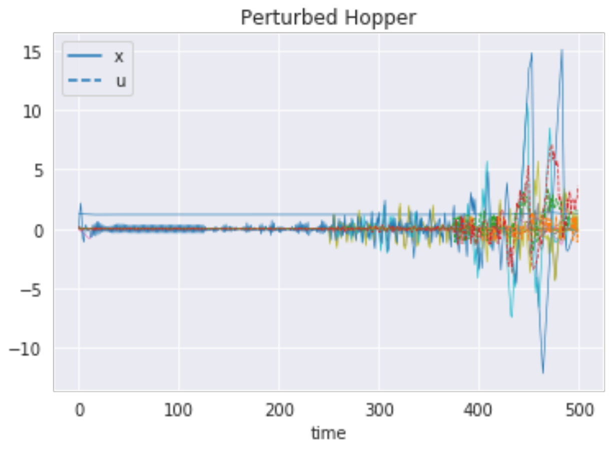
We want to keep the robots fixed upright. I've provided the fixed points, try to keep them stabilized and observe the resulting behavior (under different levels of perturbations). For example, in the hopper, we want to exhibit the behavior in the clip on the right. When running the following cells, you should see a video rendering in the notebook. The highlighted joints show which joints are being perturbed; your job is to stabilize.



```
In [11]: env = HopperModEnv()
         x_ref, u_ref = np.zeros(11), np.zeros(env.action_space.sample().shape[0])
         hopper_config = {
             'env': env,
             'f': env.f_sim,
             'exp_name': "Perturbed Hopper",
             'steps': 500,
             'x_ref': x_ref,
             'u_ref': u_ref,
             'ss':  np.array([[np.concatenate([env.init_qpos[1:],env.init_qvel])]]),
             'p_val': [0, .1, 1, 10]
         }
         lqr_nonlinear(hopper_config)
```

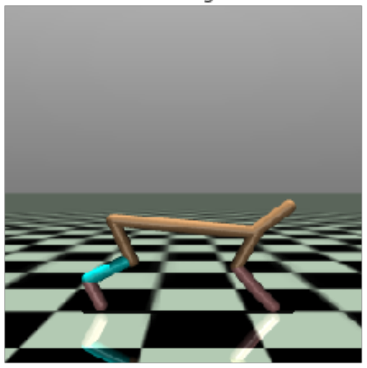
Perturbation Magnitude 10



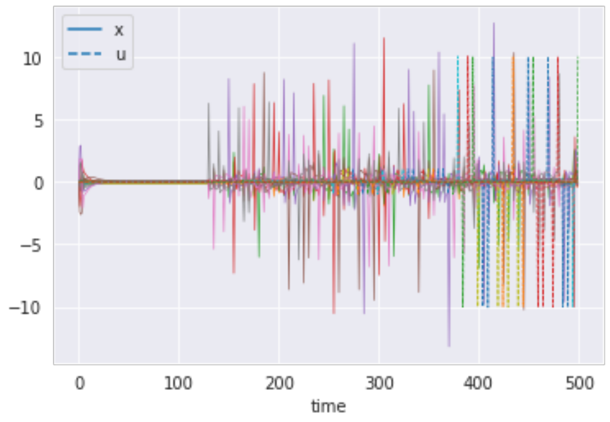


In [12]:
```
env = CheetahModEnv()
# here's a fixed point I generated using a little magic:
u_ref = np.zeros(env.action_space.sample().shape[0])
x_ref = np.array([-1.40206503e-02, -1.30077635e-01,  4.98998886e-02,  3.3471220
3e-02
                  ,6.45586857e-02, -1.40762470e-02, -5.68323106e-02, -1.3102632
7e-01
                 ,-1.21501338e-01,  1.38522957e-05, -6.43600095e-05,  2.8304223
8e-05
                  ,1.45850734e-04,  1.18368438e-04,  1.35731406e-04, -1.7116101
7e-04
                 ,-1.57742919e-04, -1.54620176e-04])

cheetah_config = {
    'env': env,
    'f': env.f_sim,
    'exp_name': "Perturbed Cheetah",
    'steps': 500,
    'x_ref': x_ref,
    'u_ref': u_ref,
    'ss':  np.array([[np.concatenate([env.init_qpos[:],env.init_qvel])]]),
    'p_val': [0, 1, 10, 100]
}
lqr_nonlinear(cheetah_config)
```

Perturbation Magnitude 100



Perturbed Cheetah



(c) Linear Time Varying (LTV) Systems

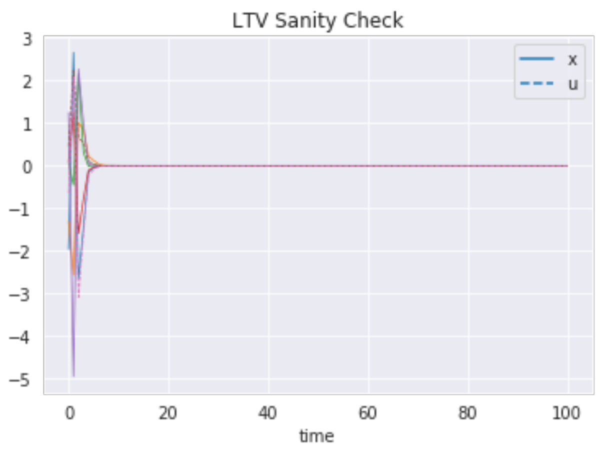
Now we'll consider LTV systems, i.e., $x_{t+1} = A_t x_t + B_t u_t$ (and Q, R may be time-dependent). Our optimal control problem is thus:

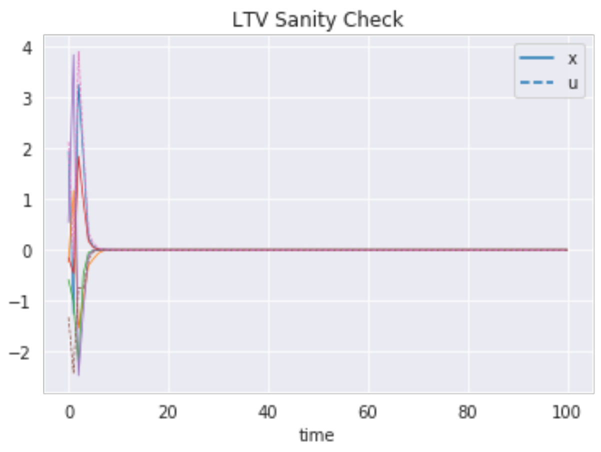
$$\min_{x,u} \sum_{t=1}^{T} (x_t' Q_t x_t + u_t' R_t u_t) + x_{T+1}' Q_T x_{T+1}$$
$$\text{s.t. } x_{t+1} = A_t x_t + B_t u_t$$

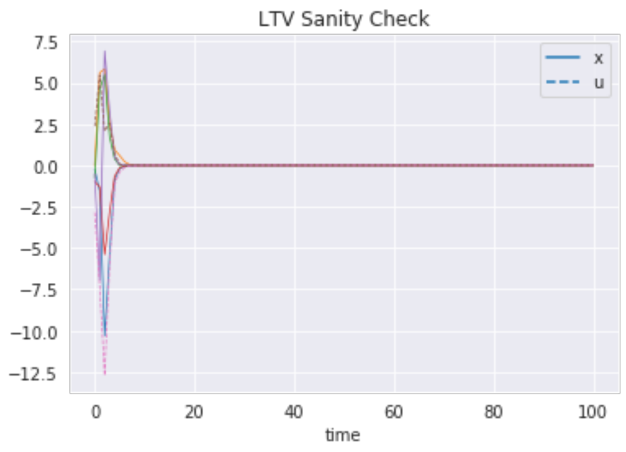
Eyeball your plots to sanity check your implementation as this will be important for part (d)!

In [13]:
```
# implement a finite horizon optimal feedback controller, accounting for possib
ly time-varying parameters
def lqr_finite_horizon(A_lst, B_lst, Q_lst, R_lst, T):
    """
    Each of A_lst, B_lst, Q_lst, and R_lst is either a python list (of length
T) of numpy arrays
        or a numpy array (indicating this parameter is not time-varying).
    You will need to handle both cases in your implementation

    Find the finite horizon K and P through running LQR back-ups
    return: K_{1:T}, P_{1:T}
    """

    K_lst, P_lst= [], []
    """YOUR CODE HERE"""
    if type(A_lst) is np.ndarray:
        return lqr_infinite_horizon(A_lst, B_lst, Q_lst, R_lst)

    P = np.eye(A_lst[0].shape[0])
    for i in range(T):
        A, B, Q, R = A_lst[-i-1], B_lst[-i-1], Q_lst, R_lst
        K = - np.linalg.inv(R + B.T @ P @ B) @ B.T @ P @ A
        K_lst.append(K)
        P = Q + K.T @ R @ K + (A + B @ K).T @ P @ (A + B @ K)
        P_lst.append(P)
    """YOUR CODE ENDS HERE"""
    return K_lst, P_lst
```

In [14]:
```
# here we define a LTV system for a fixed horizon
T = 100
A_lst = [np.array([[np.sin(t), -0.5049, 0.0299, 2.6544, 1.0608],
              [2.3846, -0.2312, -0.1260, -0.7945, 0.5279],
              [1.4019, -0.6394, -0.1401, 0.5484, 0.1624],
              [-0.0254, 0.4595, -0.0862, 2.1750, 1.1012],
              [0.5172, 0.5060, 1.6579, -0.9407, -1.4441]]) for t in range(T)]
B_lst = [np.array([[-0.7789, -1.2076],
              [0.4299, -1.6041],
              [0.2006, -1.7395],
              [0.8302, 0.2295],
              [-1.8465, np.cos(t)]]) for t in range(T)]

starting_states = np.array([[-1.9613, 1.9277, -0.2442],
                            [-1.3127, -0.2406, -0.0260],
                            [0.0698, -0.5860, -0.7522],
                            [0.0935, -0.1524, -0.9680],
                            [1.2494, 0.5397, -0.5146]])
n_starting_states = starting_states.shape[1]

dx, du = A_lst[0].shape[0], B_lst[0].shape[1]
Q, R = np.eye(dx), np.eye(du)
K_lst, P_lst = lqr_finite_horizon(A_lst, B_lst, Q, R, T)

# fill in to use your controller
def simulate(A_lst, B_lst, K_list, n_starting_states, T, noise=None):
    for s in np.arange(n_starting_states):
        x, u = np.zeros((K_list[0].shape[1], T+1)), np.zeros((K_list[0].shape
[0], T+1))
        x[:,0] = starting_states[:,s]
        for t in np.arange(T):
            """YOUR CODE HERE"""
            u[:,t] = K_list[-t - 1] @ x[:, t]
            """YOUR CODE ENDS HERE"""
            x[:,t+1] = A_lst[t] @ x[:,t] + B_lst[t] @ u[:,t]
            if noise is not None:
                x[:,t+1] += noise[:,t]

        plt.plot(x.T, linewidth=.7)
        plt.plot(np.squeeze(u.T), linewidth=.7, linestyle='--')
        legend_elements = [Line2D([0], [0], label='x'),Line2D([0], [0], linesty
le='--', label='u')]
        plt.legend(handles=legend_elements)
        plt.xlabel('time')
        plt.title("LTV Sanity Check")
        plt.show()

# simulate to sanity check your TV solution
simulate(A_lst, B_lst, K_lst, n_starting_states, T)
```


LTV Sanity Check
x
u
time


LTV Sanity Check
x
u
time


LTV Sanity Check
x
u
time

(d) Trajectory Following for Nonlinear Systems

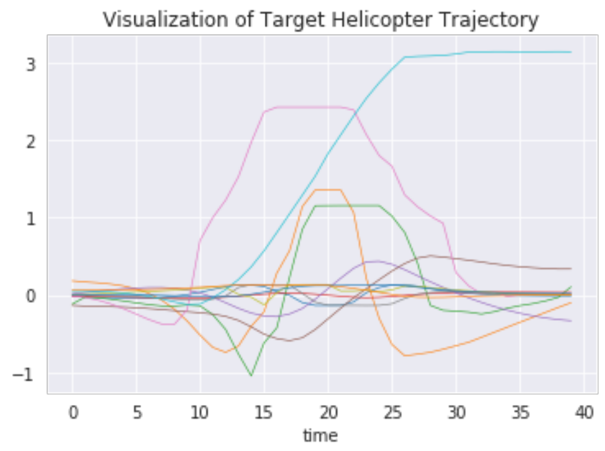
Given a feasible trajectory $\{x_t, u_t\}_{t=0}^{H-1}$, we define our optimization problem as follows:

$$\min_{u_0,\ldots,u_{H-1}} \sum_{t=0}^{H-1}(((x_t - x_t^*)'Q_t(x_t - x_t^*) + (u_t - u_t^*)'R_t(u_t - u_t^*)))$$
$$\text{s.t. } x_{t+1} = f(x_t, u_t)$$

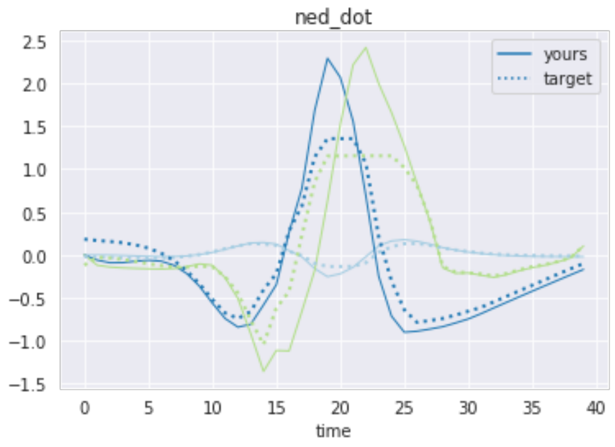
Your task is to implement trajectory following for helicopter flight (non-linear system) by **transforming the objective into a LTV setting and running LQR.** We have provided (and loaded) the reference trajectory below, run the following cell to visualize the target trajectory. **Note that this trajectory is *approximately feasible*, so you will have to include an offset term to account for this.**

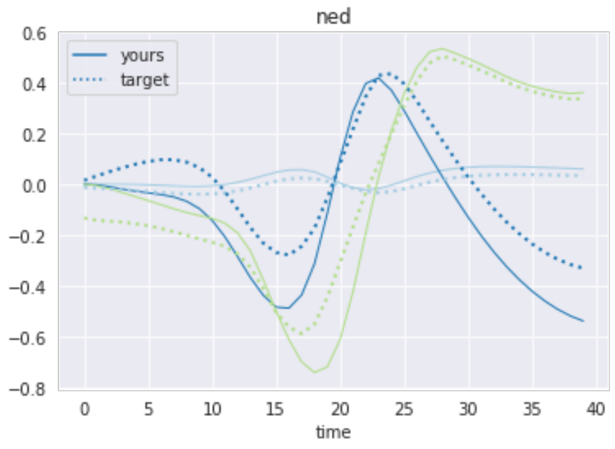
HINT: for the offset, refer to the lecture and now use the optional argument to the `linearize_dynamics` function. Now, what to do with these offsets? Since we have written a time-varying LQR solver for linear systems, in order to use the same code, augment the "state" to include the offset in your A and B matrices.

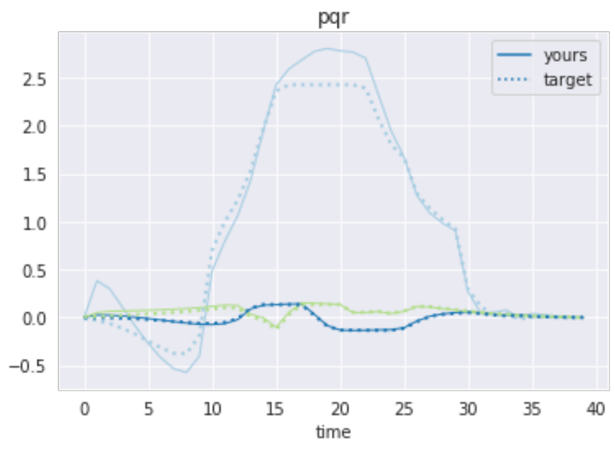
In [15]:
```
traj = loadmat("mats/heli_traj.mat")
x_init, x_target, u_target = traj['x_init'], traj['x_target'], traj['u_target']

plt.plot(x_target.T, linewidth = .6)
plt.title("Visualization of Target Helicopter Trajectory")
plt.xlabel("time")
plt.show()
```

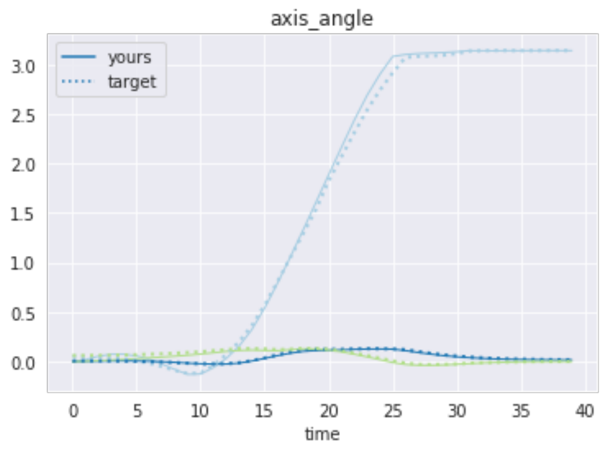


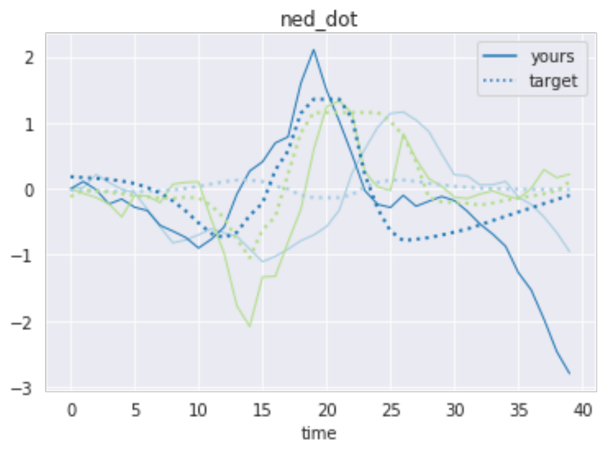
In [16]:
```
f = sim_heli
dt = 0.1 # we work with discrete time

x_ref, u_ref = x_target.T, u_target.T
my_eps = 0.001 # finite difference for numerical differentiation
T, dx = x_ref.shape
du = u_ref.shape[1]
A_lst, B_lst = [], [] # this should look familiar, maybe your code from part
(c) will be helpful!

for t in range(T-1):
    """YOUR CODE HERE"""
    A, B, c = linearize_dynamics(f, x_ref[t], u_ref[t], dt, my_eps, x_ref_tplus
1=x_ref[t+1])
    A_t = np.eye(dx + 1)
    A_t[:-1, :-1] = A
    A_t[:-1, -1] = c
    B_t = np.zeros((dx + 1, du))
    B_t[:-1, :] = B
    """YOUR CODE ENDS HERE"""
    A_lst.append(A_t)
    B_lst.append(B_t)

Q, R = np.eye(A_lst[0].shape[0]), np.eye(B_lst[0].shape[1])
Q[-1, -1] = 0
K_list, P_list = lqr_finite_horizon(A_lst, B_lst, Q, R, T-1) # you wrote this i
n part (c)

# once again fill in the control input based on your controller
def simulate(K_lst, f, x_ref, u_ref, dt, n_starting_states, T, noise=None):
    def setup_heli_idx():
        idx = dict()
        k = 0
        keys = ["ned_dot", "ned", "pqr", "axis_angle"]
        for ky in range(len(keys)):
            idx[keys[ky]] = np.arange(k,k+3)
            k += 3
        return idx
    idx = setup_heli_idx()

    def disp(sim, ref, label):
        cp = sns.color_palette("Paired")
        a, b = sim[idx[label]], ref[idx[label]]
        [plt.plot(a[i], linewidth=1, color=cp[i]) for i in range(a.shape[0])]
        [plt.plot(b[i], linewidth=2, linestyle=':', color=cp[i]) for i in range
(b.shape[0])]
        legend_elements = [Line2D([0], [0], label='yours'),Line2D([0], [0], lin
estyle=':', label='target')]
        plt.legend(handles=legend_elements)
        plt.xlabel('time')
        plt.title(label)
        plt.show()

    for s in np.arange(n_starting_states):
        x, u = np.zeros((x_ref.shape[1], T)), np.zeros((u_ref.shape[1], T))
        x[:,0] = starting_states[:,s]
        for t in np.arange(T-1):
            """YOUR CODE HERE"""
            y = x[:, t] - x_ref[t]
            z = np.ones(x.shape[0] + 1)
            z[:-1] = y
            u[:,t] = u_ref[t] + K_list[-t-1] @ z
            """YOUR CODE ENDS HERE"""
```

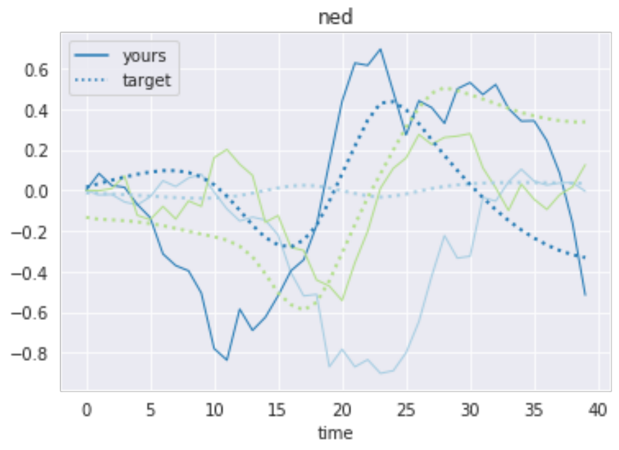

ned_dot
yours
target
time

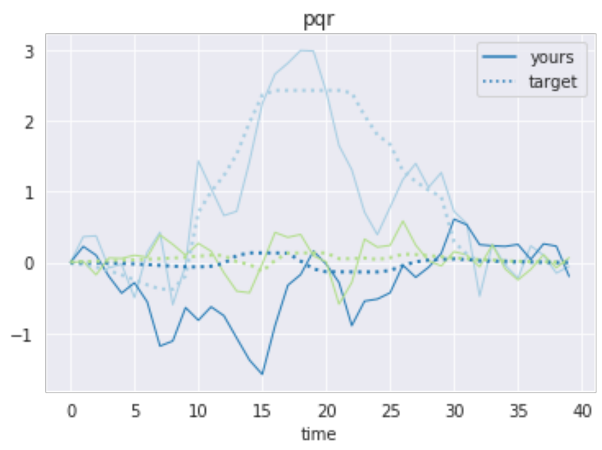

ned
yours
target
time

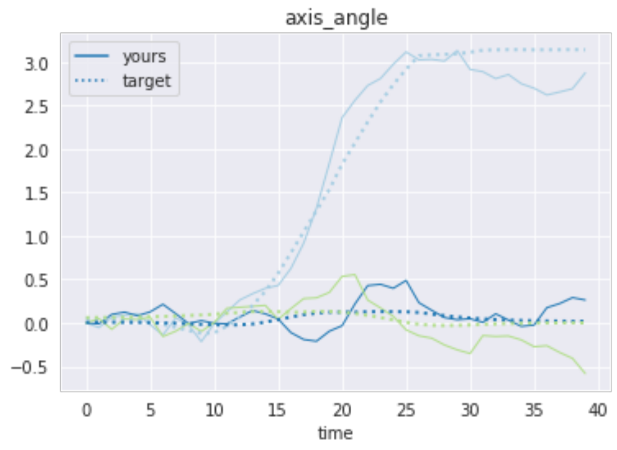

pqr
yours
target
time


axis_angle
yours
target
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
5
10
15
20
25
30
35
40
time


ned_dot
yours
target
2
1
0
−1
−2
−3
0
5
10
15
20
25
30
35
40
time


ned
yours
target
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
0
5
10
15
20
25
30
35
40
time





Congratulations! You're finished with the programming exercises. Print preview -> print to pdf then include the pdf in your write up.

Remember to complete the theory questions for the write-up as well!