Spring 2022

Stanford
AA 203: Optimal and Learning-based Control
Homework #2, due May 2 by 11:59 pm

Pei-Chen Wu pcwu1023
Collaborators: Kevin Lee and Albert Chan

Problem 1: Shortest path through a grid

Consider the shortest path problem in Figure 1 where it is only possible to travel to the right and the numbers represent the travel times for each segment. The control input is the decision to go "up-right" or "down-right" at each node.

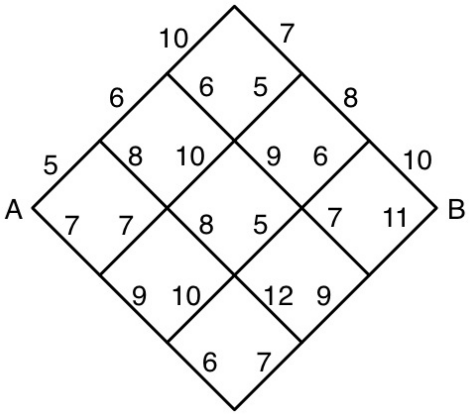


Figure 1: Shortest path problem on a grid.

(a) Use Dynamic Programming (DP) to find the shortest path from A to B.

ANSWER:
$J_N^*(B) = 0$

$J_{N-1}^*(O) = 10 + J_N^*(B) = 10;\ u_{N-1}^*(O) = DR$
$J_{N-1}^*(P) = 11 + J_N^*(B) = 11;\ u_{N-1}^*(P) = UR$

$J_{N-2}^*(L) = 8 + J_{N-1}^*(O) = 18;\ u_{N-2}^*(L) = DR$
$J_{N-2}^*(M) = min(6 + J_{N-1}^*(O), 7 + J_{N-1}^*(P)) = 16;\ u_{N-2}^*(M) = UR$
$J_{N-2}^*(N) = 9 + J_{N-1}^*(P) = 20;\ u_{N-2}^*(N) = UR$

$J_{N-3}^*(H) = 7 + J_{N-2}^*(L) = 25;\ u_{N-3}^*(H) = DR$
$J_{N-3}^*(I) = min(5 + J_{N-2}^*(L), 9 + J_{N-2}^*(M)) = 23;\ u_{N-3}^*(I) = UR$
$J_{N-3}^*(J) = min(5 + J_{N-2}^*(M), 12 + J_{N-2}^*(N)) = 21;\ u_{N-3}^*(J) = UR$
$J_{N-3}^*(K) = 7 + J_{N-2}^*(N) = 27;\ u_{N-3}^*(K) = UR$

$J^*_{N-4}(E) = min(10 +^*_{N-3} (H), 6 + J^*_{N-3}(I)) = 29;\ u^*_{N-4}(E) = DR$
$J^*_{N-4}(F) = min(10 + J^*_{N-3}(I), 8 + J^*_{N-3}(J))) = 29;\ u^*_{N-4}(F) = DR$
$J^*_{N-4}(G) = min(10 + J^*_{N-3}(J), 6 + J^*_{N-3}(K)) = 31;\ u^*_{N-4}(G) = UR$

$J^*_{N-5}(C) = min(6 +^*_{N-4} (E), 8 + J^*_{N-4}(F)) = 35;\ u^*_{N-5}(C) = UR$
$J^*_{N-5}(D) = min(7 + J^*_{N-4}(F), 9 + J^*_{N-4}(G))) = 36;\ u^*_{N-5}(D) = UR$

$J^*_{N-6}(A) = min(5 + J^*_{N-5}(C), 7 + J^*_{N-5}(D))) = 40;\ u^*_{N-6}(A) = UR$

The shortest path is UR, UR, DR, UR, DR, DR.

(b) Consider a generalized version of the shortest path problem in Figure 1 where the grid has n segments on each side. Find the number of computations required by an exhaustive search algorithm (i.e., the number of routes that such an algorithm would need to evaluate) and the number of computations required by a DP algorithm (i.e., the number of DP evaluations). For example, for $n = 3$ as in Figure 1, an exhaustive search algorithm requires 20 computations, while the DP algorithm requires only 15.

ANSWER:
Exhaustive search algorithm:
For finding a feasible path, we have to always go up right n times in total and down right n times in total. Total number of moves is $2n$. Therefore, we can randomly select any n moves to be up right or down right and we don't need to worry about the permutation of it, so the total number of the exhaustive search algorithm requires C_n^{2n}

DP algorithm:
DP algorithm needs to compute cost at each node, therefore, the total number requires to compute will be the total number of nodes, which is $(n+1)^2 - 1 = n^2 + 2n$

Problem 2: Machine maintenance

Suppose we have a machine that is either running or is broken down. If it runs throughout one week, it makes a gross profit of \$100. If it fails during the week, gross profit is zero. If it is running at the start of the week and we perform preventive maintenance, the probability that it will fail during the week is 0.4. If we do not perform such maintenance, the probability of failure is 0.7. However, maintenance will cost \$20. When the machine is broken down at the start of the week, it may either be repaired at a cost of \$40, in which case it will fail during the week with a probability of 0.4, or it may be replaced at a cost of \$150 by a new machine that is guaranteed to run through its first week of operation. Find the optimal repair, replacement, and maintenance policy that maximizes total profit over four weeks, assuming a new machine at the start of the first week.

ANSWER:
Applying DP and start from the end of forth week.

$J_4^*(run) = 0$
$J_4^*(down) = 0$

$J_3^*(run) = max_{u \in Maintenance, NoAction}(0.6 \times (-20 + 100) + 0.4 \times (-20), 0.3 \times 100 + 0.7 \times 0) = 40;$
$u_3^*(run) = Maintenance$
$J_3^*(down) = max_{u \in Repair, Replace}(0.6 \times (-40 + 100) + 0.4 \times (-40), 1 \times (-50)) = 20;$
$u_3^*(down) = Repair$

$J_2^*(run) = max_{u \in Maintenance, NoAction}(0.6 \times (-20 + 100 + 40) + 0.4 \times (-20 + 20), 0.3 \times (100 + 40) + 0.7 \times (0 + 20)) = 72;$
$u_2^*(run) = Maintenance$
$J_2^*(down) = max_{u \in Repair, Replace}(0.6 \times (-40 + 100 + 40) + 0.4 \times (-40 + 20), 1 \times (-50 + 40)) = 52;$
$u_2^*(down) = Repair$

$J_1^*(run) = max_{u \in Maintenance, NoAction}(0.6 \times (-20 + 100 + 72) + 0.4 \times (-20 + 52), 0.3 \times (100 + 72) + 0.7 \times (0 + 52)) = 104;$
$u_1^*(run) = Maintenance$

$J_0^*(run) = 1 \times (100 + 104) = 204$

The optimal policy is when the machine is running we have to do maintenance, if the machine is down, we have to do repair.

Problem 3: Markovian drone

In this problem, we will apply techniques for solving a Markov Decision Process (MDP) to guide a flying drone to its destination through a storm. The world is represented as an $n \times n$ grid, i.e., the state space is

$$\mathcal{S} \coloneqq \{(x_1, x_2) \in \mathbb{R}^2 \mid x_1, x_2 \in \{0, 1 \ldots, n-1\}\}.$$

In these coordinates, $(0, 0)$ represents the bottom left corner of the map and $(n-1, n-1)$ represents the top right corner of the map. From any location $x = (x_1, x_2) \in \mathcal{S}$, the drone has four possible directions it can move in, i.e.,

$$\mathcal{A} \coloneqq \{\texttt{up}, \texttt{down}, \texttt{left}, \texttt{right}\}.$$

The corresponding state changes for each action are:

- `up`: $(x_1, x_2) \mapsto (x_1, x_2 + 1)$
- `down`: $(x_1, x_2) \mapsto (x_1, x_2 - 1)$
- `left`: $(x_1, x_2) \mapsto (x_1 - 1, x_2)$
- `right`: $(x_1, x_2) \mapsto (x_1 + 1, x_2)$

Additionally, there is a storm centered at $x_{\text{eye}} \in \mathcal{S}$. The storm's influence is strongest at its center and decays farther from the center according to the equation $\omega(x) = \exp\left(-\frac{\|x - x_{\text{eye}}\|_2^2}{2\sigma^2}\right)$. Given its current state x and action a, the drone's next state is determined as follows:

- With probability $\omega(x)$, the storm will cause the drone to move in a uniformly random direction.
- With probability $1 - \omega(x)$, the drone will move in the direction specified by the action.
- If the resulting movement would cause the drone to leave $\mathcal{S}$, then it will not move at all. For example, if the drone is on the right boundary of the map, then moving right will do nothing.

The quadrotor's objective is to reach $x_{\text{goal}} \in \mathcal{S}$, so the reward function is the indicator function $R(x) = I_{x_{\text{goal}}}(x)$. In other words, the drone will receive a reward of 1 if it reaches the $x_{\text{goal}} \in \mathcal{S}$, and a reward of 0 otherwise. The reward of a trajectory in this infinite horizon problem is a discounted sum of the rewards earned in each timestep, with discount factor $\gamma \in (0, 1)$.

(a) Given $n = 20$, $\sigma = 10$, $\gamma = 0.95$, $x_{\text{eye}} = (15, 15)$, and $x_{\text{goal}} = (19, 9)$, write code that uses value iteration to find the optimal value function for the drone to navigate the storm. Recall that value iteration repeats the Bellman update

$$V(x) \leftarrow \max_{a \in \mathcal{A}} \left(\sum_{x' \in \mathcal{S}} p(x'; x, a)(R(x') + \gamma V(x')) \right)$$

until convergence, where $p(x'; x, a)$ is the probability distribution of the next state being x' after taking action a in state x, and R is the reward function. Plot a heatmap of the optimal value function obtained by value iteration over the grid $\mathcal{S}$, with $x = (0, 0)$ in the bottom left corner, $x = (n-1, n-1)$ in the top right corner, the x_1-axis along the bottom edge, and the x_2-axis along the left edge.

ANSWER:
CODE: Attached at the end.

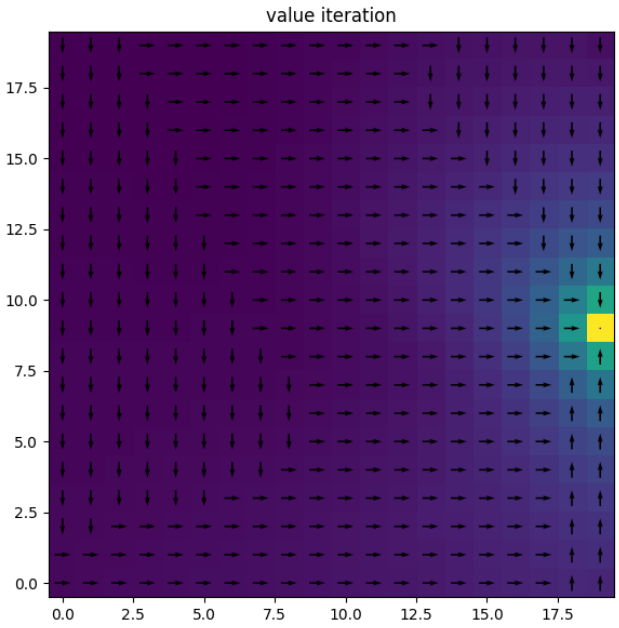


Figure 2: Heatmap of optimal values (color) and optimal policy (arrow)

(b) Recall that a policy π is a mapping $\pi : \mathcal{S} \to \mathcal{A}$ where $\pi(x)$ specifies the action to be taken should the drone find itself in state x. An optimal value function V^* induces an optimal policy π^* such that

$$\pi^*(x) \in \arg\max_{a \in \mathcal{A}} \left(\sum_{x' \in \mathcal{S}} p(x'; x, a)(R(x') + \gamma V^*(x')) \right)$$

Use the value function you computed in part (a) to compute an optimal policy. Then, use this policy to simulate the MDP starting from over $N = 100$ time steps starting at $x = (0, 19)$. Plot the policy as a heatmap where the actions $\{\texttt{up}, \texttt{down}, \texttt{left}, \texttt{right}\}$ correspond to the values $\{0, 1, 2, 3\}$, respectively. Plot the simulated drone trajectory overlaid on the policy heatmap, and briefly describe in words what the policy is doing.

CODE: Attached at the end.
ANSWER:
From part (a), we can get optimal value of each state, so from the current state, the drone can go up, down, left and right. We can know that which direction can give us the maximum value, which will be the optimal policy for the drone to move. We can see the optimal policy don't have any go left option, most of the optimal policies are either down or right, which can make drone avoid storm and also go towards the goal ASAP. We can see that from the stochastic drone trajectory it goes all the way down and tries to go right and up to reach the goal, which obviously shows that the drone is trying to avoid the storm eye at $(15, 15)$. The drone will not necessary follow the optimal policy is because from the problem statement there is a probability $w(x)$ which causes the drone to move in a uniformly random direction. After $N = 100$ iteration, the drone is moving around the goal, since we don't terminate the drone when it reaches the goal, therefore, because of stochasticity, it's moving around the goal even after it reaches the goal.

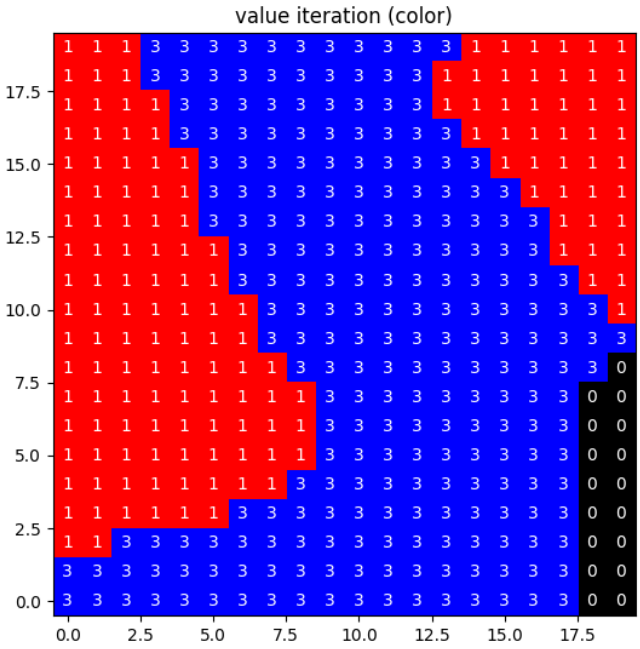


Figure 3: Optimal Policy

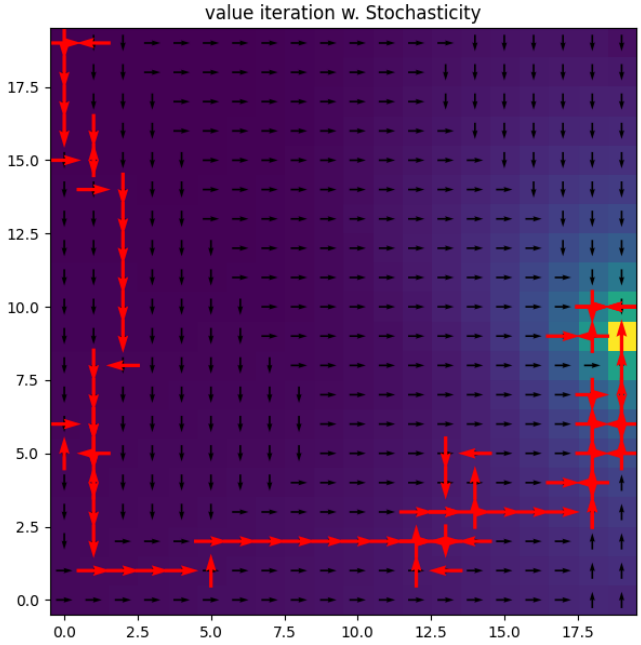


Figure 4: Drone Trajectory w. Stochasticity

Problem 4: Cart-pole balance

In this problem, we will design a controller to balance an inverted pendulum on a cart, i.e., the classic "cart-pole" benchmark. This system has two degrees of freedom corresponding to the horizontal position x of the cart, and the angle θ of the pendulum (where $\theta = 0$ occurs when the pendulum is handing straight down). We can apply a force $u \in \mathbb{R}$ to push the cart horizontally, where $u > 0$ corresponds to a force in the positive x-direction. With the state $s \coloneqq (x, \theta, \dot{x}, \dot{\theta}) \in \mathbb{R}^4$, we can write the continuous-time dynamics of the cart-pole system as

$$\dot{s} = f(s,u) = \begin{bmatrix} \dot{x} \\ \dot{\theta} \\ \frac{m_p(\ell\dot{\theta}^2 + g\cos\theta)\sin\theta + u}{m_c + m_p\sin^2\theta} \\ -\frac{(m_c+m_p)g\sin\theta + m_p\ell\dot{\theta}^2\sin\theta\cos\theta + u\cos\theta}{\ell(m_c+m_p\sin^2\theta)} \end{bmatrix},$$

where m_p is the mass of the pendulum, m_c is the mass of the cart, ℓ is the length of the pendulum, and g is the acceleration due to gravity. We can discretize the continuous-time dynamics using Euler integration with a fixed time step Δt to get the approximate discrete-time dynamics

$$s_{k+1} \approx s_k + \Delta t f(s_k, u_k),$$

where s_k and u_k are the state and control input, respectively, at time $t = k\Delta t$.

(a) Consider the upright state $s^* \coloneqq (0, \pi, 0, 0)$ with $u^* \coloneqq 0$, and define $\Delta s_k \coloneqq s_k - s^*$. Linearizing the approximate discrete-time dynamics $s_{k+1} \approx s_k + \Delta t f(s_k, u_k)$ about (s^*, u^*) yields an approximate LTI system of the form

$$\Delta s_{k+1} \approx A\Delta s_k + Bu_k.$$

Express A and B in terms of m_p, m_c, ℓ, g, and Δt. You may use the fact that

$$\frac{\partial f}{\partial s}(s^*, u^*) = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & \frac{m_p g}{m_c} & 0 & 0 \\ 0 & \frac{(m_c+m_p)g}{m_c\ell} & 0 & 0 \end{bmatrix}, \quad \frac{\partial f}{\partial u}(s^*, u^*) = \begin{bmatrix} 0 \\ 0 \\ \frac{1}{m_c} \\ \frac{1}{m_c\ell} \end{bmatrix}.$$

ANSWER:

$$s_{k+1} \approx s_k + \Delta t f(s_k, u_k)$$

$$s_{k+1} \coloneqq \Delta s_{k+1} + s^*_{k+1}$$

$$s_k \coloneqq \Delta s_k + s^*_k$$

$$\Delta s_{k+1} + s^*_{k+1} \approx \Delta s_k + s^*_k + \Delta t f(s_k, u_k)$$

$$f(s_k, u_k) \approx f(s_k^*, u_k^*) + \frac{\partial f}{\partial s}(s_k^*, u_k^*)(s_k - s^*) + \frac{\partial f}{\partial u}(s_k^*, u_k^*)(u_k - u^*)$$

$u^* = 0$,

$$f(s_k, u_k) \approx f(s_k^*, u_k^*) + \frac{\partial f}{\partial s}(s_k^*, u_k^*)\Delta s_k + \frac{\partial f}{\partial u}(s_k^*, u_k^*)u_k$$

$$\Delta s_{k+1} + s_{k+1}^* \approx \Delta s_k + s_k^* + \Delta t f(s_k^*, u_k^*) + \Delta t \frac{\partial f}{\partial s}(s_k^*, u_k^*)\Delta s_k + \Delta t \frac{\partial f}{\partial u}(s_k^*, u_k^*)u_k$$

Since $s_{k+1}^* = s_k^* + \Delta t f(s_k^*, u_k^*)$. We can cancel out s_{k+1}^* for both sides.

$$\Delta s_{k+1} \approx \Delta s_k + \Delta t \frac{\partial f}{\partial s}(s_k^*, u_k^*)\Delta s_k + \Delta t \frac{\partial f}{\partial u}(s_k^*, u_k^*)u_k$$

$$\Delta s_{k+1} \approx (I + \Delta t \frac{\partial f}{\partial s}(s^*, u^*))\Delta s_k + \Delta t \frac{\partial f}{\partial u}(s^*, u^*)u_k$$

Therefore, we can get A and B in the following expression. A and B don't depend on the state variables, and A and B only depend on time delta, gravity, masses and length of the cart-pole.

$$A = I + \Delta t \frac{\partial f}{\partial s}(s^*, u^*)$$

$$B = \Delta t \frac{\partial f}{\partial u}(s^*, u^*)$$

We will design a stabilizing LQR controller for this discrete-time LTI system to solve

$$\begin{aligned} &\underset{\{u_k\}_{k=0}^{\infty}}{\text{minimize}} \quad \sum_{k=0}^{\infty}\left(\frac{1}{2}\Delta s_k^{\mathsf{T}} Q \Delta s_k + \frac{1}{2} u_k^{\mathsf{T}} R u_k\right) \\ &\text{subject to} \quad \Delta s_{k+1} = A\Delta s_k + B u_k, \ \forall k \in \mathbb{N}_{\geq 0} \end{aligned},$$

for fixed $Q, R \succ 0$. Recall that after N iterations of the discrete-time Riccati recursion

$$\begin{aligned} K_k &= -(R + B^{\mathsf{T}} P_{k+1} B)^{-1} B^{\mathsf{T}} P_{k+1} A \\ P_k &= Q + A^{\mathsf{T}} P_{k+1}(A + B K_k) \end{aligned},$$

the cost-to-go matrices $\{P_k\}_{k=0}^{N}$ and the time-varying feedback gains $\{K_k\}_{k=0}^{N-1}$ describe the optimal LQR controller for a finite-horizon version of the problem above. If (A, B) is stabilizable, then these iterates asymptotically converge to some $P_\infty \succ 0$ and K_∞, such that $(s_0 - s^*)^{\mathsf{T}} P_\infty (s_0 - s^*) > 0$ is the finite optimal cost-to-go for any initialization s_0, and $u_k = K_\infty \Delta s_k$ is the *time-invariant* feedback law for the corresponding optimal LQR controller[1].

(b) Write code to approximate P_∞ and K_∞ for the linearized, discretized cart-pole system by initializing $P_\infty = 0$ and then applying the Ricatti recursion until convergence with respect to the maximum element-wise norm condition $\|P_{k+1} - P_k\|_{\max} < 10^{-4}$. Use $m_p = 2$ kg, $m_c = 10$ kg, $\ell = 1$ m, $g = 9.81$ m/s^2, $\Delta t = 0.1$ s, $Q = I_4$, and $R = I_1$. Report the value of K_∞ up to two decimal places for each entry.

CODE: Attached at the end.
ANSWER:
K = array([[0.73, -231.85, 4.22, -68.25]])

[1]The infinite-horizon LQR problem also converges for fixed $Q \succeq 0$ and $R \succ 0$, as long as (A, B) is stabilizable and (A, Q) is observable.

(c) Write code to simulate the continuous-time, nonlinear cart-pole system with the linear feedback controller $u = K_\infty \Delta s$. Initialize the system at $s = (0, 3\pi/4, 0, 0)$, and use a controller sampling rate of 10 Hz. Plot each state variable over time on separate plots for $t \in [0, 30]$. For your own interest, we provide the function `animate_cartpole` in `animations.py` to create a video animation of the cart-pole over time[2].

Hint: Write a function `ds = cartpole(s,t,u)` that computes the state derivative `ds` for the continuous-time, nonlinear cart-pole dynamics. To simulate the cart-pole with the fixed control input `u[k]` from state `s[k]` at time `t[k]` to state `s[k+1]` at time `t[k+1]`, you can use the following Python code:

```
from scipy.integrate import odeint
s[k+1] = odeint(cartpole, s[k], t[k:k+2], (u[k],))[1]
```

Make sure to review the documentation for `odeint`.

CODE: Attached at the end.

Plots:

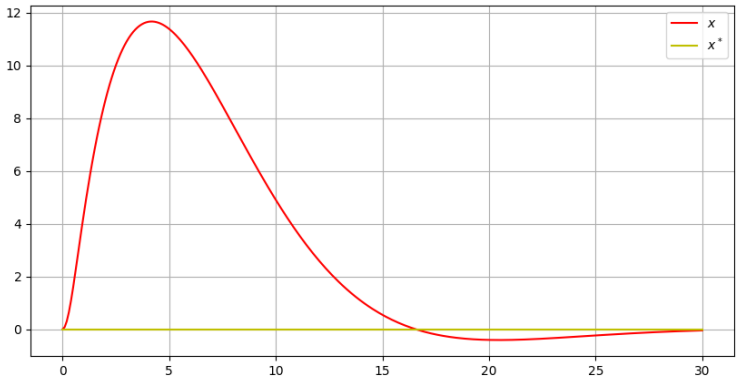


Figure 5: x vs. time

[2]See https://github.com/StanfordASL/AA203-Homework.

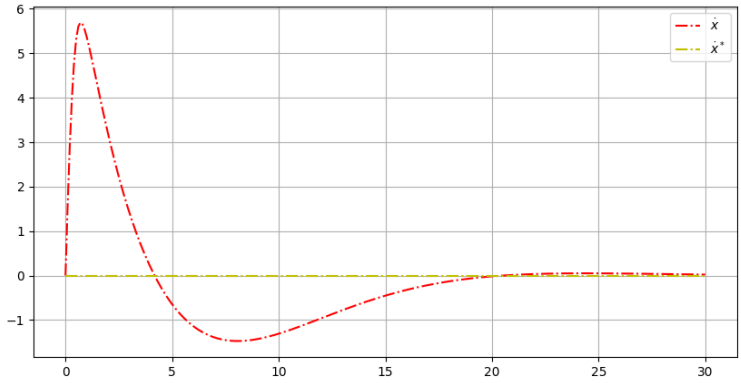


Figure 6: $\dot{x}$ vs. time

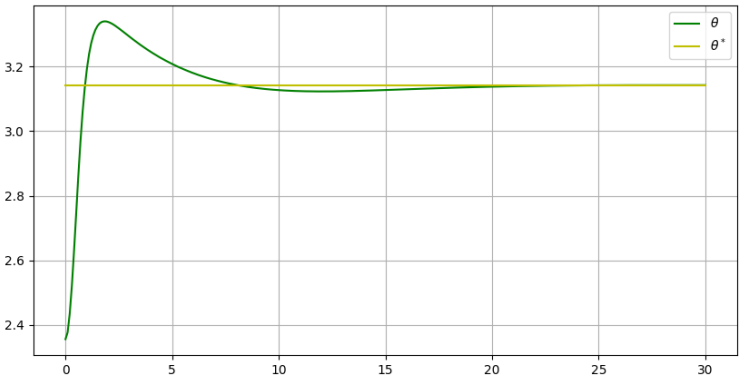


Figure 7: θ vs. time

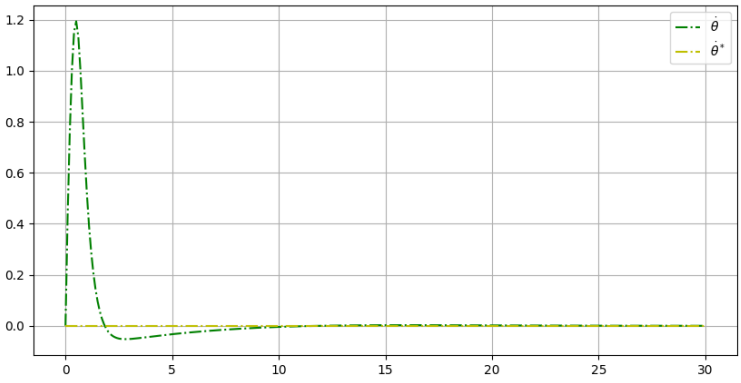


Figure 8: $\dot{\theta}$ vs. time

(d) To investigate the disturbance rejection ability of the controller, add noise to the system dynamics. Specifically, after each controller sampling period (i.e., every 0.1 s), sample a new noise vector $w \in \mathbb{R}^4$ from the Gaussian distribution with mean $\mu = 0$ and covariance $\Sigma = \text{diag}(0, 0, 10^{-3}, 10^{-3})$, and add it to the state. Simulate the noisy system and plot each state variable over time for $t \in [0, 30]$.

CODE: Attached at the end.
Plots:

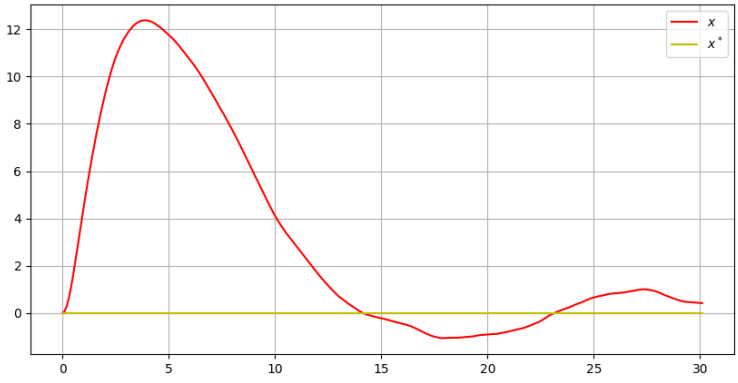


Figure 9: x w. noise vs. time

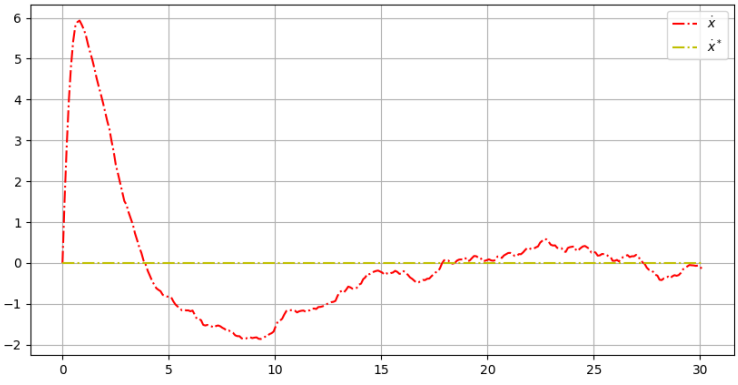


Figure 10: $\dot{x}$ w. noise vs. time

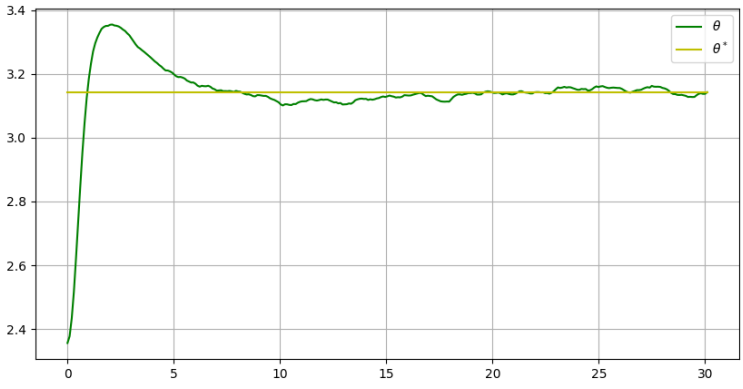


Figure 11: θ w. noise vs. time

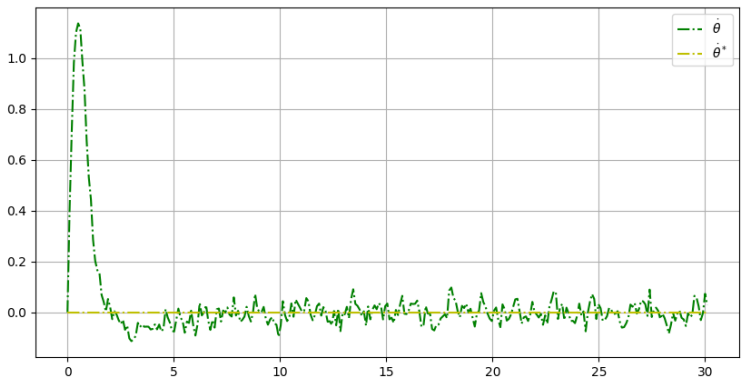


Figure 12: $\dot{\theta}$ w. noise vs. time

(e) We will now use an LQR controller to track a time-varying trajectory. Specifically, we will aim to balance the pendulum upright (i.e., $\theta^*(t) \equiv \pi$) while oscillating the position of the cart to track a desired reference $x^*(t) = a\sin(2\pi t/T)$, where $a > 0$ and $T > 0$ are known constants.

i. Normally, as derived in class when applying LQR for trajectory tracking, you would have to re-linearize the system around the desired trajectory at each time step; why is this not the case for this particular problem (i.e., why can you just reuse A and B)?

ANSWER:
From part (a), we can see that x, $\dot{x}$, θ and $\dot{\theta}$ won't affect A and B. When we do linearization at s^*, the $f(s^*, u^*)$ has been cancelled out when linearization and A and B don't depend on the state variables, it only depends on time delta, gravity, masses and length of cart-pole. Therefore, reuse A and B will not be a problem.

ii. Repeat part (c) (i.e., without noise) for this case with $a = 10$ and $T = 30$, except this time initialize the system upright at $s(0) = (0, \pi, 0, 0)$. For each state plot, overlay the corresponding entry from the reference trajectory $s^*(t)$.
CODE: Attached at the end.
Plots:

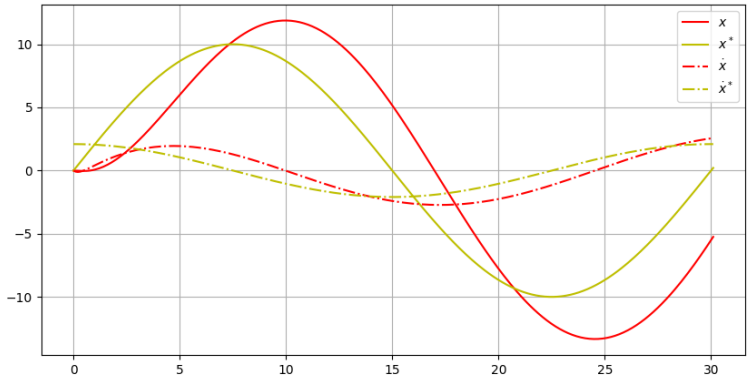


Figure 13: x vs. time $Q = I$

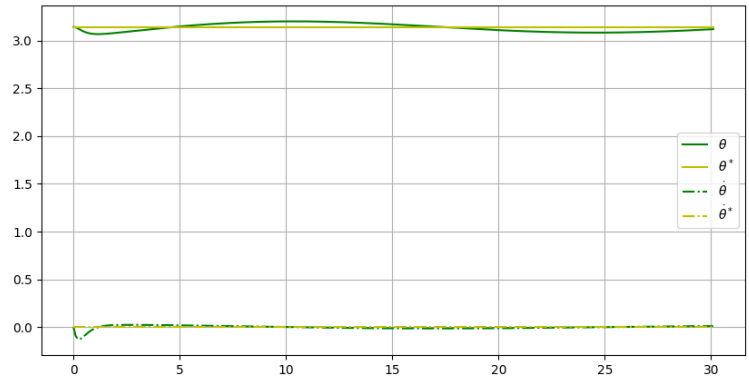


Figure 14: θ vs. time $Q = I$

iii. You may notice that this controller does not have good tracking performance. You could try increasing the state penalty matrix Q to, e.g., $Q = 10I_4$. However, this should only improve tracking for x and $\dot{x}$, while $\theta(t)$ and $\dot{\theta}(t)$ still oscillate around zero. What physical characteristic of the desired trajectory (or lack thereof) causes this to happen? CODE: Attached at the end.

ANSWER:
We can observe that increase Q will also increase our controller gain K matrix, which will decrease the steady state error. However, we cannot fully eliminate the steady state error with simply increase values of K matrix. The reason why θ is oscillating around π is the cart-pole itself is oscillating around $x = 0$, since it's trajectory is sine wave, which if the cart tracks sine wave trajectory perfectly, the pendulum has to oscillate around π.

Plots:

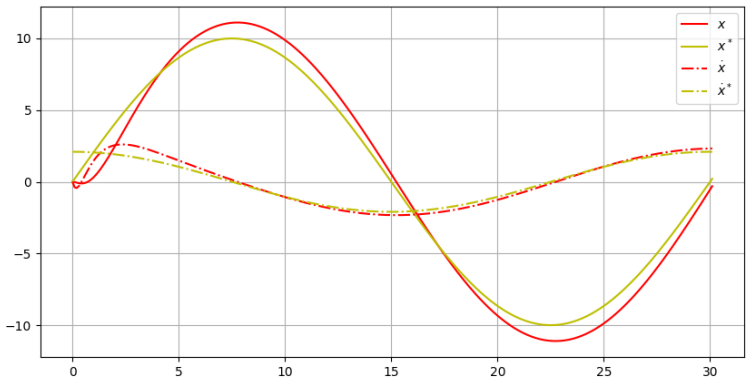


Figure 15: x vs. time $Q = 100I$

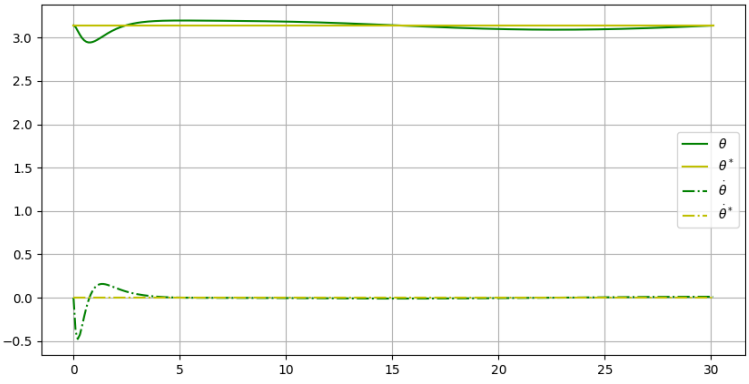


Figure 16: θ vs. time $Q = 100I$

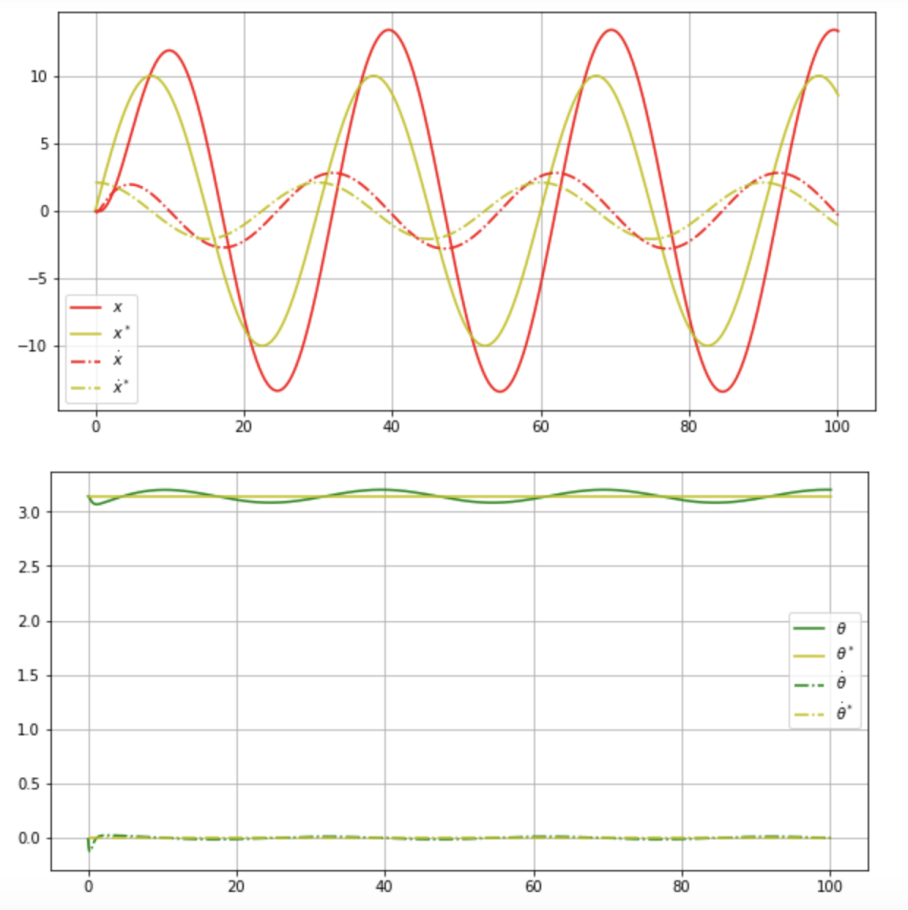


Figure 17: x vs. time $Q = I$ 100sec

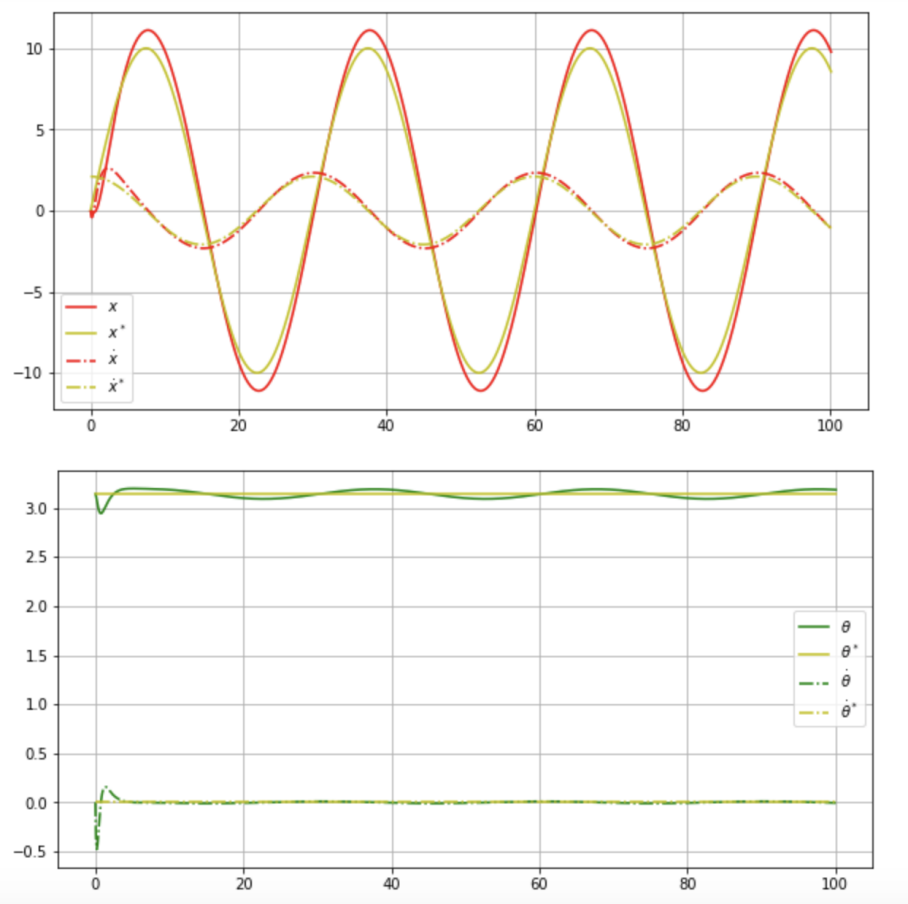


Figure 18: θ vs. time $Q = 100I$ 100sec

Learning goals for this problem set:

Problem 1: To familiarize with the DP algorithm and to appreciate the computational savings of DP versus an exhaustive search algorithm.

Problem 2: To apply dynamic programming in stochastic environments by reasoning about expected utilities.

Problem 3: To solve a stochastic optimization problem with value iteration by formulating it as an MDP.

Problem 4: To gain experience with implementing LQR controllers by coding them "from scratch".

In [1]:
```
import numpy as np, tensorflow as tf, matplotlib.pyplot as plt
```

In [2]:
```
n = 20
m = n
sdim, adim = m * n, 4

# the parameters of the storm
x_eye, sig = np.array([15, 15]), 10

x_goal = (19,19)

w_fn = lambda x: np.exp(-((np.linalg.norm(np.array(x) - x_eye)**2)/(2*sig**2)))
```

In [3]:
```
def helpful_matrices():
    y, x = np.meshgrid(np.arange(n), np.arange(m))
    idx2pos = np.stack([x.reshape(-1), y.reshape(-1)], -1)
    pos2idx = np.reshape(np.arange(m * n), (m, n))
    return idx2pos, pos2idx
```

In [4]:
```
def make_transition_matrices():

    pt_min, pt_max = [0, 0], [m - 1, n - 1]

    idx2pos, pos2idx = helpful_matrices()

    T_right, T_up, T_left, T_down = [np.zeros((sdim, sdim)) for _ in range(4)]

    for i in range(m):
        for j in range(n):
            w = w_fn(np.array([i,j])) # w at current pt

            right = np.clip(np.array([i,j]) + np.array([1, 0]), pt_min, pt_max)
            up = np.clip(np.array([i,j]) + np.array([0, 1]), pt_min, pt_max)
            left = np.clip(np.array([i,j]) + np.array([-1, 0]), pt_min, pt_max)
            down = np.clip(np.array([i,j]) + np.array([0, -1]), pt_min, pt_max)

            # specify action = right
            T_right[pos2idx[i, j], pos2idx[right[0], right[1]]] = 1 - w
            T_right[pos2idx[i, j], pos2idx[up[0], up[1]]] = w / 3
            T_right[pos2idx[i, j], pos2idx[left[0], left[1]]] = w / 3
            T_right[pos2idx[i, j], pos2idx[down[0], down[1]]] = w / 3

            # specify action = up
            T_up[pos2idx[i, j], pos2idx[right[0], right[1]]] = w / 3
            T_up[pos2idx[i, j], pos2idx[up[0], up[1]]] = 1 - w
            T_up[pos2idx[i, j], pos2idx[left[0], left[1]]] = w / 3
            T_up[pos2idx[i, j], pos2idx[down[0], down[1]]] = w / 3

            # specify action = left
            T_left[pos2idx[i, j], pos2idx[right[0], right[1]]] = w / 3
            T_left[pos2idx[i, j], pos2idx[up[0], up[1]]] = w / 3
            T_left[pos2idx[i, j], pos2idx[left[0], left[1]]] = 1 - w
            T_left[pos2idx[i, j], pos2idx[down[0], down[1]]] = w / 3

            # specify action = down
            T_down[pos2idx[i, j], pos2idx[right[0], right[1]]] = w / 3
            T_down[pos2idx[i, j], pos2idx[up[0], up[1]]] = w / 3
            T_down[pos2idx[i, j], pos2idx[left[0], left[1]]] = w / 3
            T_down[pos2idx[i, j], pos2idx[down[0], down[1]]] = 1 - w

    return pos2idx, idx2pos, (T_right, T_up, T_left, T_down)
```

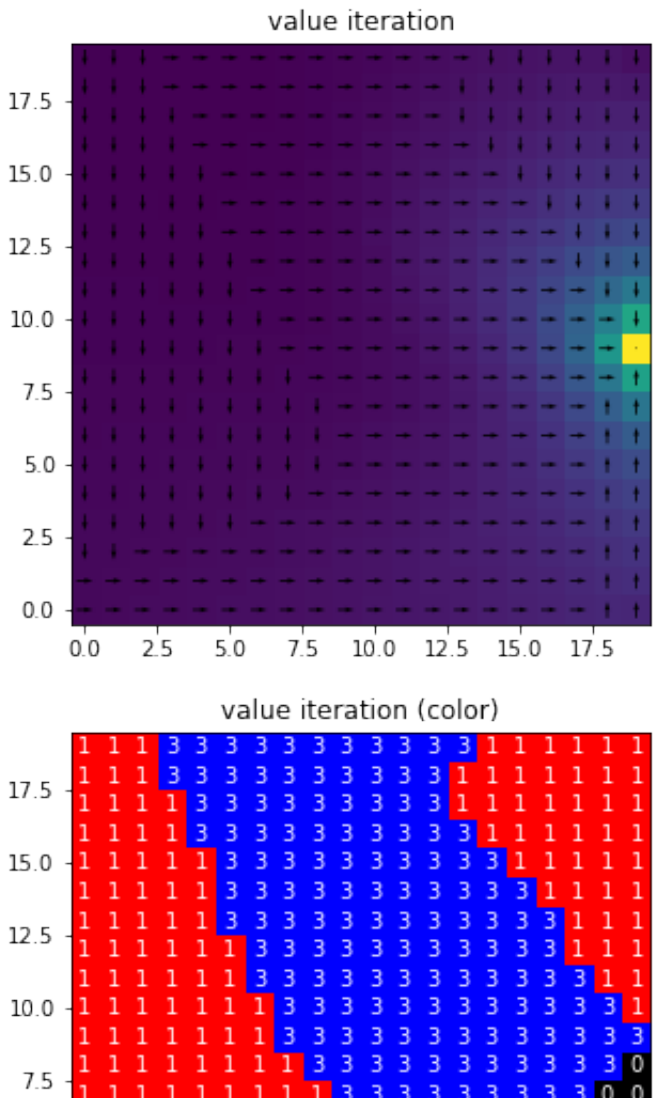
In [5]:
```
def value_iteration(Ts, reward, mask, gam):

    V = tf.zeros([sdim])

    # perform value iteration
    for _ in range(1000):
        V_stack = tf.stack([V,V,V,V],-1)
        V_stack = tf.transpose(V_stack, perm=[1,0])
        V_prev = V

        V = tf.math.reduce_max(reward + tf.reshape(gam*(Ts@V_stack[:,:,None])*mask[None,:,None],[4,400]), axis=0)

        err = tf.linalg.norm(V-V_prev)

        if err < 1e-8:
            break

    return V
```

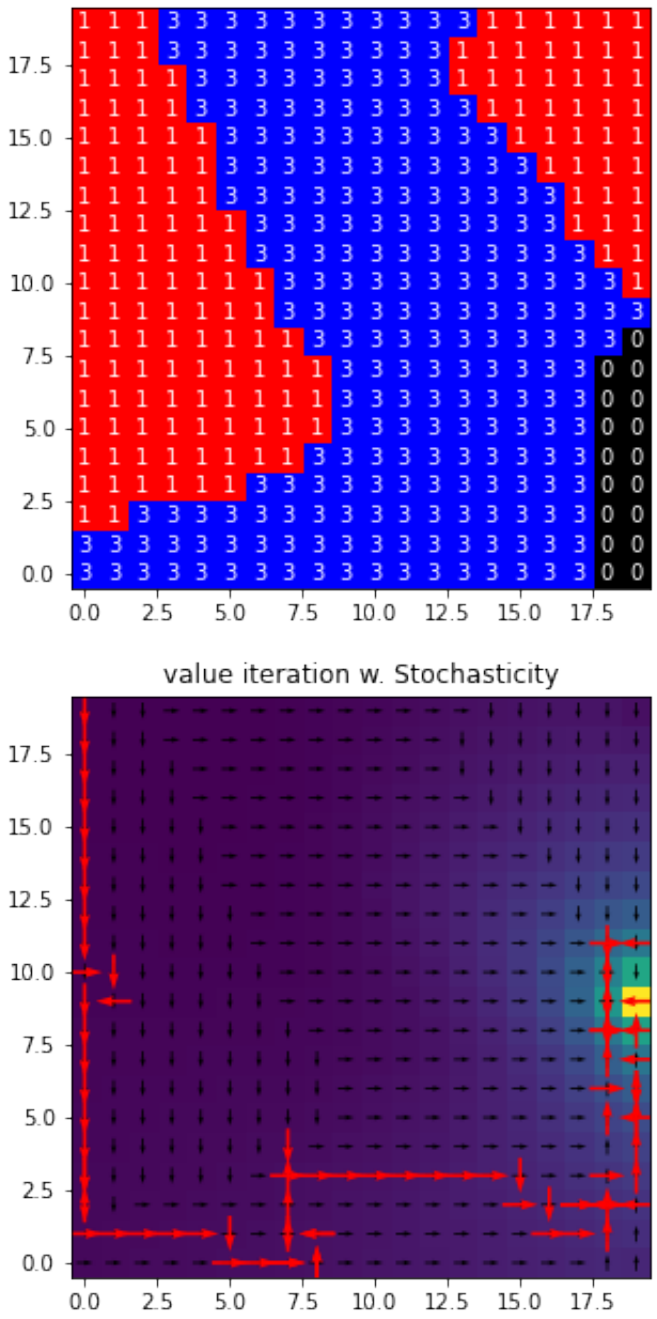
In [6]:
```
def visualize_value_function(V):
    pt_min, pt_max = [0, 0], [m - 1, n - 1]
    V = np.array(V)

    X, Y = np.meshgrid(np.arange(m), np.arange(n))
    pts = np.stack([X.reshape(-1), Y.reshape(-1)], -1)

    u, v = [], []
    for pt in pts:
        pt_right = np.clip(np.array(pt) + np.array([1, 0]), pt_min, pt_max)
        pt_up = np.clip(np.array(pt) + np.array([0, 1]), pt_min, pt_max)
        pt_left = np.clip(np.array(pt) + np.array([-1, 0]), pt_min, pt_max)
        pt_down = np.clip(np.array(pt) + np.array([0, -1]), pt_min, pt_max)
        next_pts = [pt_up, pt_down, pt_left, pt_right]
        Vs = [V[next_pt[0], next_pt[1]] for next_pt in next_pts]
        idx = np.argmax(Vs)
        u.append(next_pts[idx][0] - pt[0])
        v.append(next_pts[idx][1] - pt[1])
    u, v = np.reshape(u, (m, n)), np.reshape(v, (m, n))

    plt.imshow(V.T, origin="lower")
    plt.quiver(X, Y, u, v, pivot="middle")
```

In [7]:
```
def visualize_value_function_color(V):
    pt_min, pt_max = [0, 0], [m - 1, n - 1]
    V = np.array(V)

    X, Y = np.meshgrid(np.arange(m), np.arange(n))
    pts = np.stack([X.reshape(-1), Y.reshape(-1)], -1)

    u, v, opt_act_list = [], [], []
    for pt in pts:
        pt_right = np.clip(np.array(pt) + np.array([1, 0]), pt_min, pt_max)
        pt_up = np.clip(np.array(pt) + np.array([0, 1]), pt_min, pt_max)
        pt_left = np.clip(np.array(pt) + np.array([-1, 0]), pt_min, pt_max)
        pt_down = np.clip(np.array(pt) + np.array([0, -1]), pt_min, pt_max)
        next_pts = [pt_up, pt_down, pt_left, pt_right]
        Vs = [V[next_pt[0], next_pt[1]] for next_pt in next_pts]
        idx = np.argmax(Vs)
        opt_act_list.append(idx)
        u.append(next_pts[idx][0] - pt[0])
        v.append(next_pts[idx][1] - pt[1])
    u, v, opt_act = np.reshape(u, (m, n)), np.reshape(v, (m, n)), np.reshape(opt_act_list, (m, n))

    palette = np.array([[  0,   0,   0],   # black
                        [255,   0,   0],   # red
                        [  0, 255,   0],   # green
                        [  0,   0, 255]])   # blue

    for i in range(20):
        for j in range(20):
            text = plt.text(j, i, opt_act[i,j], ha="center", va="center", color='w')


    plt.imshow(palette[opt_act], origin="lower")
```

In [8]:
```
def visualize_value_function_with_Stochasticity(V):
    pt_min, pt_max = [0, 0], [m - 1, n - 1]
    V = np.array(V)

    X, Y = np.meshgrid(np.arange(m), np.arange(n))
    pts = np.stack([X.reshape(-1), Y.reshape(-1)], -1)

    u, v = [], []
    for pt in pts:
        pt_right = np.clip(np.array(pt) + np.array([1, 0]), pt_min, pt_max)
        pt_up = np.clip(np.array(pt) + np.array([0, 1]), pt_min, pt_max)
        pt_left = np.clip(np.array(pt) + np.array([-1, 0]), pt_min, pt_max)
        pt_down = np.clip(np.array(pt) + np.array([0, -1]), pt_min, pt_max)
        next_pts = [pt_right, pt_up, pt_left, pt_down]
        Vs = [V[next_pt[0], next_pt[1]] for next_pt in next_pts]
        idx = np.argmax(Vs)
        u.append(next_pts[idx][0] - pt[0])
        v.append(next_pts[idx][1] - pt[1])

    u, v = np.reshape(u, (m, n)), np.reshape(v, (m, n))

    plt.imshow(V.T, origin="lower")
    plt.quiver(X, Y, u, v, pivot="middle")


    u, v = [], []
    optimal_policy = [pts[380]] # start point (0,19)
    pt = pts[380]
    for _ in range(100):
        pt_right = np.clip(np.array(pt) + np.array([1, 0]), pt_min, pt_max)
        pt_up = np.clip(np.array(pt) + np.array([0, 1]), pt_min, pt_max)
        pt_left = np.clip(np.array(pt) + np.array([-1, 0]), pt_min, pt_max)
        pt_down = np.clip(np.array(pt) + np.array([0, -1]), pt_min, pt_max)
        next_pts = [pt_right, pt_up, pt_left, pt_down]
        Vs = [V[next_pt[0], next_pt[1]] for next_pt in next_pts]
        idx = np.argmax(Vs)

        rd = np.random.uniform(0,1) # with w probabality, the storm will cause the drone to move in a uniformly random directio
        if rd < w_fn(pt):
            idx = int(np.random.choice(np.delete(np.array([0,1,2,3]), idx), size=1))

        u.append(next_pts[idx][0] - pt[0])
        v.append(next_pts[idx][1] - pt[1])

        optimal_policy.append(np.array([next_pts[idx][0],next_pts[idx][1]]))
        pt = next_pts[idx]

    plt.quiver(np.array(optimal_policy)[:-1,0],np.array(optimal_policy)[:-1,1], u, v, pivot="middle", color='red')
```

In [12]:
```
pos2idx, idx2pos, Ts = make_transition_matrices()

Ts = [tf.convert_to_tensor(T, dtype=tf.float32) for T in Ts]
Ts = tf.convert_to_tensor(Ts,dtype=tf.float32)

# create the terminal mask vector
mask = np.zeros([sdim])
mask[pos2idx[19, 9]] = 1.0
mask = tf.convert_to_tensor(mask, dtype=tf.float32)
mask = tf.cast(mask==0, dtype=tf.float32)

# generate the reward vector
reward = np.zeros([sdim, 4])
reward[pos2idx[19, 9], :] = 1.0
reward = tf.convert_to_tensor(reward, dtype=tf.float32)
reward = tf.transpose(reward, perm=[1,0])


gam = 0.95
V_opt = value_iteration(Ts, reward, mask, gam)

plt.figure(figsize=(5, 5))
visualize_value_function(np.array(V_opt).reshape((n, n)))
plt.title("value iteration")
plt.show()

plt.figure(figsize=(5, 5))
visualize_value_function_color(np.array(V_opt).reshape((n, n)))
plt.title("value iteration (color)")
plt.show()

plt.figure(figsize=(5, 5))
visualize_value_function_with_Stochasticity(np.array(V_opt).reshape((n, n)))
plt.title("value iteration w. Stochasticity")
plt.show()
```


value iteration
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
value iteration (color)


value iteration w. Stochasticity
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5

In [1]:

```
"""
Animations for various dynamical systems using `matplotlib`.

Author: Spencer M. Richards
        Autonomous Systems Lab (ASL), Stanford
        (GitHub: spenrich)
"""

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
import matplotlib.transforms as mtransforms
import matplotlib.animation as animation


def animate_cartpole(t, x, θ):
    """Animate the cart-pole system from given position data.

    All arguments are assumed to be 1-D NumPy arrays, where `x` and `θ` are the
    degrees of freedom of the cart-pole over time `t`.

    Example usage:
        import matplotlib.pyplot as plt
        from animations import animate_cartpole
        fig, ani = animate_cartpole(t, x, θ)
        ani.save('cartpole.mp4', writer='ffmpeg')
        plt.show()
    """
    # Geometry
    cart_width = 2.
    cart_height = 1.
    wheel_radius = 0.3
    wheel_sep = 1.
    pole_length = 5.
    mass_radius = 0.25

    # Figure and axis
    fig, ax = plt.subplots(dpi=100)
    x_min, x_max = np.min(x) - 1.1*pole_length, np.max(x) + 1.1*pole_length
    y_min = -pole_length
    y_max = 1.1*(wheel_radius + cart_height + pole_length)
    ax.plot([x_min, x_max], [0., 0.], '-', linewidth=1, color='k')[0]
    ax.set_xlim([x_min, x_max])
    ax.set_ylim([y_min, y_max])
    ax.set_yticks([])
    ax.set_aspect(1.)

    # Artists
    cart = mpatches.FancyBboxPatch((0., 0.), cart_width, cart_height,
                                   facecolor='tab:blue', edgecolor='k',
                                   boxstyle='Round,pad=0.,rounding_size=0.05')
    wheel_left = mpatches.Circle((0., 0.), wheel_radius, color='k')
    wheel_right = mpatches.Circle((0., 0.), wheel_radius, color='k')
    mass = mpatches.Circle((0., 0.), mass_radius, color='k')
    pole = ax.plot([], [], '-', linewidth=3, color='k')[0]
    trace = ax.plot([], [], '--', linewidth=2, color='tab:orange')[0]
    timestamp = ax.text(0.1, 0.9, '', transform=ax.transAxes)

    ax.add_patch(cart)
    ax.add_patch(wheel_left)
    ax.add_patch(wheel_right)
    ax.add_patch(mass)

    def animate(k, t, x, θ):
        # Geometry
        cart_corner = np.array([x[k] - cart_width/2, wheel_radius])
        wheel_left_center = np.array([x[k] - wheel_sep/2, wheel_radius])
        wheel_right_center = np.array([x[k] + wheel_sep/2, wheel_radius])
        pole_start = np.array([x[k], wheel_radius + cart_height])
        pole_end = pole_start + pole_length*np.array([np.sin(θ[k]),
                                                      -np.cos(θ[k])])

        # Cart
        cart.set_x(cart_corner[0])
        cart.set_y(cart_corner[1])

        # Wheels
        wheel_left.set_center(wheel_left_center)
        wheel_right.set_center(wheel_right_center)

        # Pendulum
        pole.set_data([pole_start[0], pole_end[0]],
                      [pole_start[1], pole_end[1]])
        mass.set_center(pole_end)
        mass_x = x[:k+1] + pole_length*np.sin(θ[:k+1])
        mass_y = wheel_radius + cart_height - pole_length*np.cos(θ[:k+1])
        trace.set_data(mass_x, mass_y)

        # Time-stamp
        timestamp.set_text('t = {:.1f} s'.format(t[k]))

        artists = (cart, wheel_left, wheel_right, pole, mass, trace, timestamp)
        return artists

    dt = t[1] - t[0]
    ani = animation.FuncAnimation(fig, animate, t.size, fargs=(t, x, θ),
```

```
                                    interval=dt*1000, blit=True)
    return fig, ani
```

In [2]:
```
"""
Cartpole AA203 HW2 P4

Author: Pei-Chen Wu
"""

from scipy.integrate import odeint
#import sympy as sp
from scipy.stats import multivariate_normal

mp = 2
mc = 10
l = 1
g = 9.81
```

In [3]:
```
def cal_PK(Q_multiply = 1):

    delta_t = 0.1
    Q = np.eye(4,4)*Q_multiply
    R = np.eye(1)

    df_ds_star = np.array([[0,0,1,0],
                           [0,0,0,1],
                           [0, mp*g/mc,0,0],
                           [0,(mc+mp)*g/(mc*l),0,0]])

    df_du_star = np.array([[0],
                           [0],
                           [1/mc],
                           [1/(mc*l)]])

    P = np.zeros((4,4)) # init

    A = (np.eye(4,4) + delta_t*df_ds_star)
    B = delta_t*df_du_star

    err = np.inf

    while err >= 1e-4:

        K = -np.linalg.inv(R+B.T@P@B)@B.T@P@A
        P_next = Q+A.T@P@(A+B@K)

        err = np.max(np.abs(P_next-P))

        P = P_next


    return K
```

In [4]:
```
def noise():
    return np.random.multivariate_normal([0.0, 0.0, 0.0, 0.0], np.diag([0,0,1e-3,1e-3]),1)
```

In [16]:
```
K = cal_PK(1)
K
```

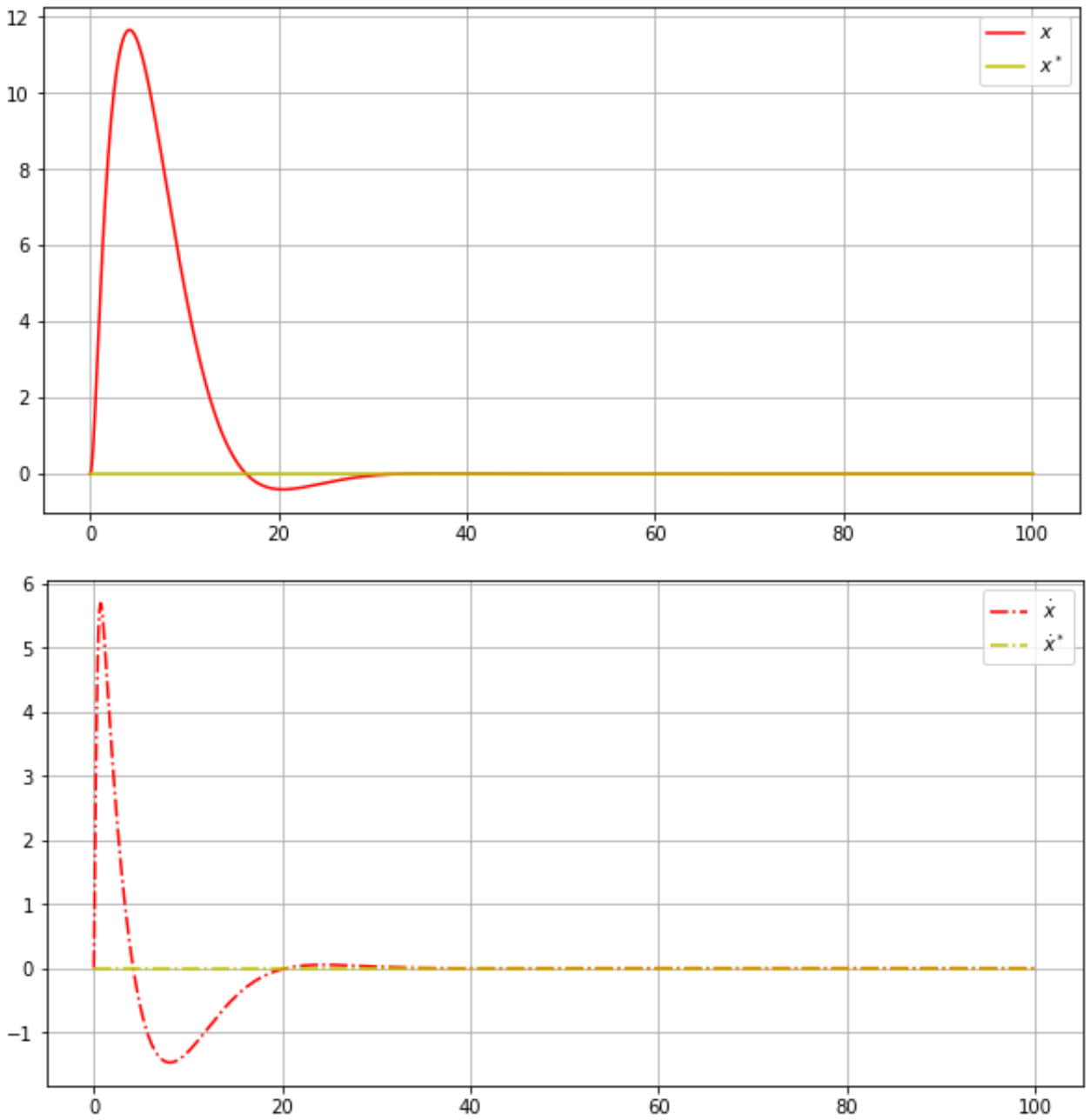
Out[16]:
```
array([[   0.7291397 , -231.85419273,    4.21967187,  -68.24742822]])
```

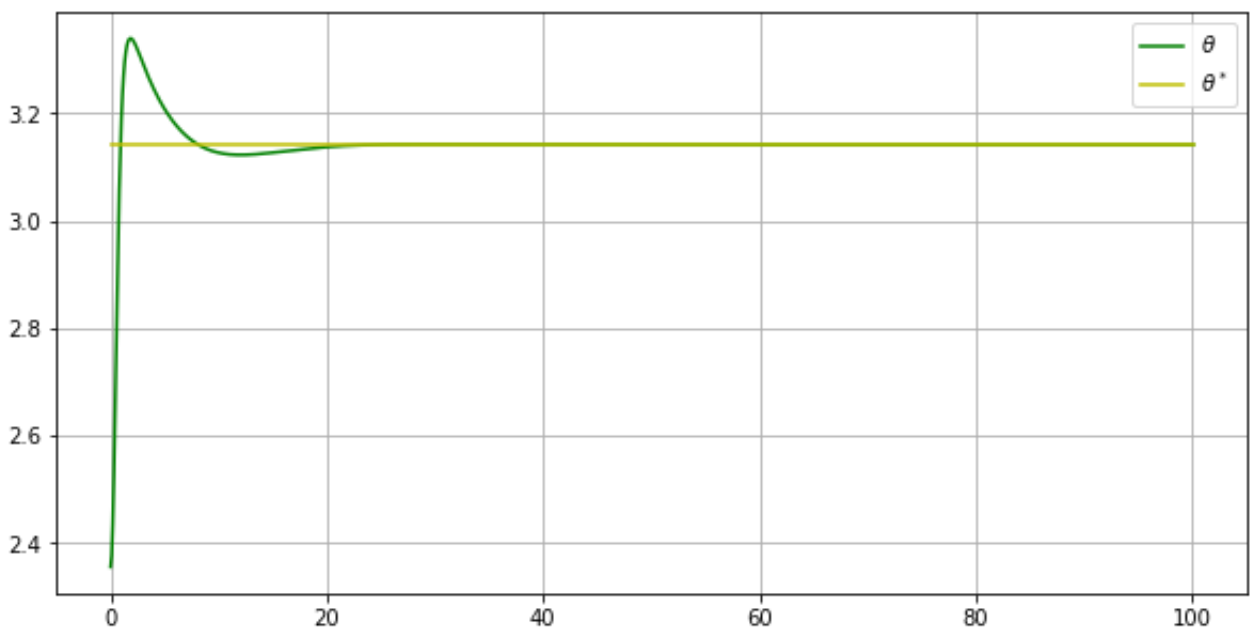
In [15]:
```
K = cal_PK(10)
K
```

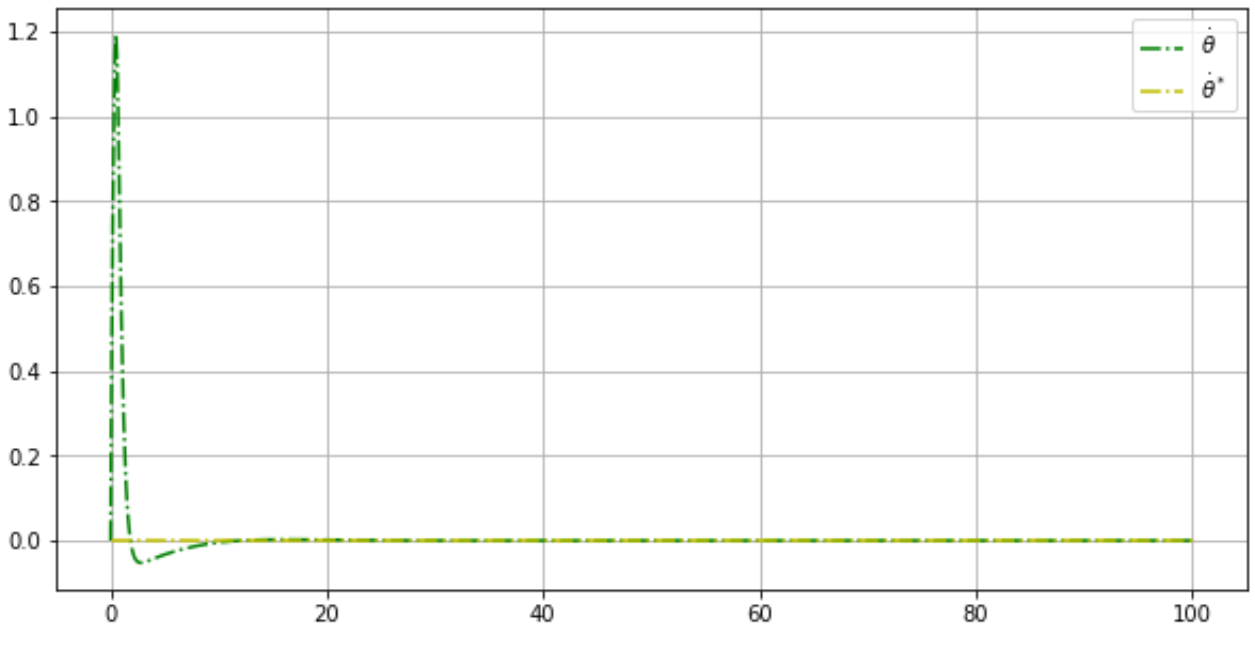
Out[15]:
```
array([[   2.26379564, -256.34294732,    8.42679164,  -76.04341843]])
```

In [19]:
```
#now make the time series
max_time = 100
freq = 10 #second subdivision (frequency/Hz)
dt = 1/freq #useful for the animation function
t = np.linspace(0,max_time+dt,int(freq*max_time)+1)

def cartpole(s,t,u):
    x = s[0]
    th = s[1]
    dx = s[2]
    dth = s[3]

    ddx = (mp*(l*dth*dth+g*np.cos(th))*np.sin(th)+u)/(mc+mp*np.sin(th)*np.sin(th))
    ddth = -1*((mc+mp)*g*np.sin(th)+mp*l*dth*dth*np.sin(th)*np.cos(th)+u*np.cos(th))/(l*(mc+mp*np.sin(th)*np.sin(th)))

    return np.array([dx, dth, ddx, ddth])

def simulate_sys(add_noise = False, part_e = False, Q_multiply = 1):

    K = cal_PK(Q_multiply)[0]

    if part_e:
        s_init = np.array([0,np.pi,0,0]) # s init
        #s_star = np.array([10*np.sin((2*np.pi/30)*0), np.pi, 10*(2*np.pi/30)*np.cos((2*np.pi/30)*0), 0])
    else:
        s_init = np.array([0,3*np.pi/4,0,0]) # s init
        s_star = np.array([0, np.pi, 0, 0])

    u = np.zeros(int(freq*max_time)+1)
    s_list = np.zeros((int(freq*max_time)+1,4))

    s_list[0,:] = s_init

    for i in range(len(t)-1):
        if part_e:
            s_star = np.array([10*np.sin(2*np.pi*t[i]/30),
                               np.pi,
                               10*(2*np.pi/30)*np.cos(2*np.pi*t[i]/30),
                               0])

        u[i] = K.dot(s_list[i,:] - s_star)
        s_list[i+1,:] = odeint(cartpole, s_list[i,:], t[i:i+2], (u[i],))[1]

        if add_noise:
            s_list[i+1,:] += noise()[0] # add noise to the system

    return s_list
```

In [20]:
```
s_t = simulate_sys(add_noise = False, part_e = False, Q_multiply=1)

#Now plot the pos vs time
plt.figure(figsize=(10, 5))
plt.grid() #this allows us to see the grid
plt.plot(t,s_t[:,0],'r',label='$x$')
plt.plot(t, np.zeros(t.shape), 'y', label='$x^*$')
plt.legend(fontsize=10) #show the legend

plt.show() #this says display the info here


#Now plot the pos vs time
plt.figure(figsize=(10, 5))
plt.grid() #this allows us to see the grid
plt.plot(t,s_t[:,2],'-.r',label='$\\dot{x}$')
plt.plot(t, np.zeros(t.shape), '-.y', label='$\\dot{x}^*$')
plt.legend(fontsize=10) #show the legend

plt.show() #this says display the info here


#Now plot the angle vs time
plt.figure(figsize=(10, 5))
plt.grid() #this allows us to see the grid
plt.plot(t,s_t[:,1],'g',label='$\\theta$')
plt.plot(t,np.ones(t.shape)*np.pi,'y',label='$\\theta^*$')
plt.legend(fontsize=10) #show the legend

plt.show() #this says display the info here


#Now plot the angle vs time
plt.figure(figsize=(10, 5))
plt.grid() #this allows us to see the grid
plt.plot(t,s_t[:,3],'-.g',label='$\\dot{\\theta}$')
plt.plot(t,np.zeros(t.shape),'-.y',label='$\\dot{\\theta}^*$')
plt.legend(fontsize=10) #show the legend

plt.show() #this says display the info here
```

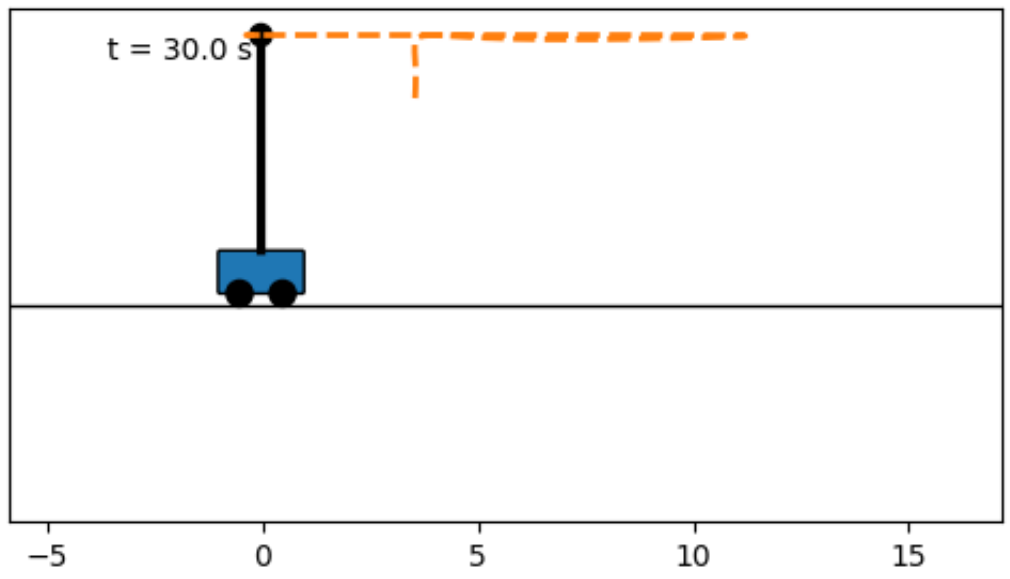






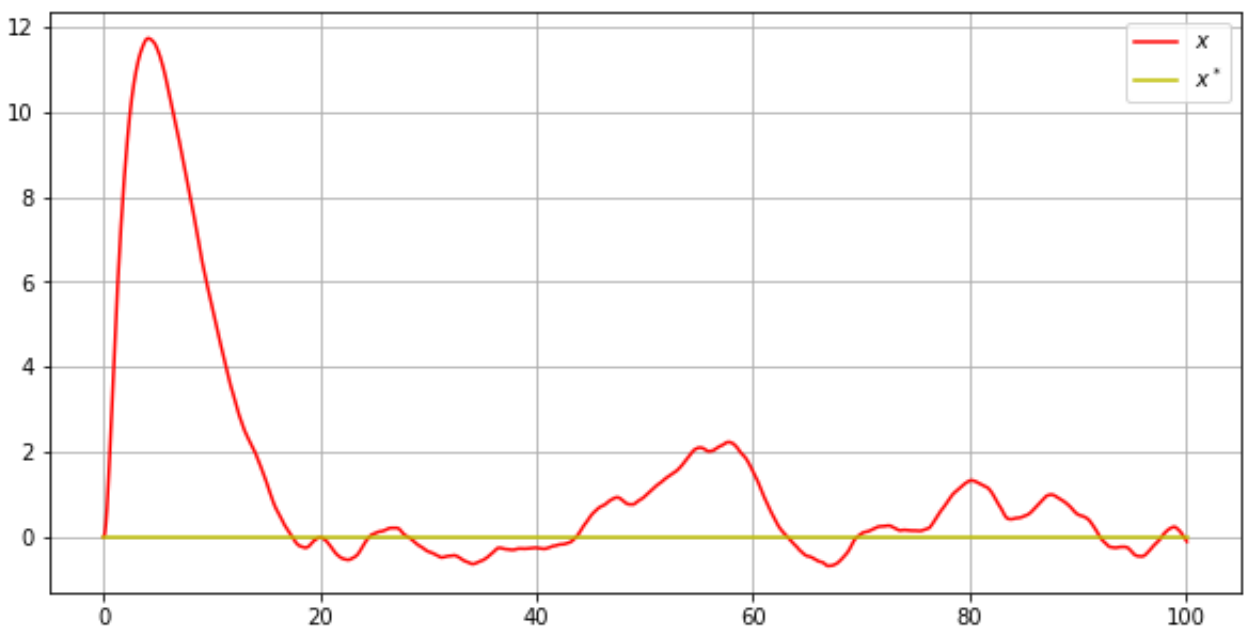
In [8]:

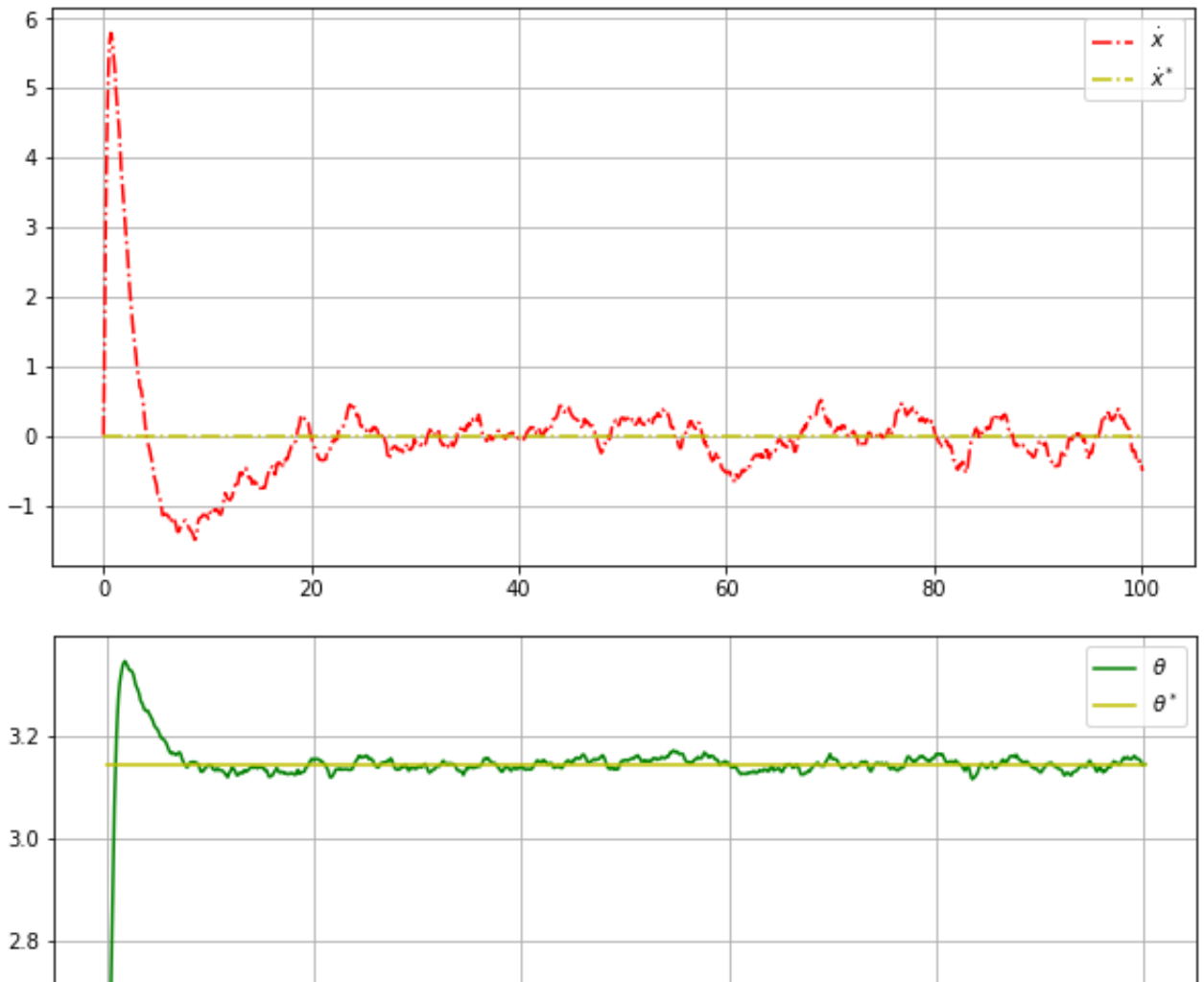
```
x = s_t[:,0]
θ = s_t[:,1]
fig, ani = animate_cartpole(t[:-1], x, θ)
ani.save('cartpole_c.mp4', writer='ffmpeg')
plt.show()
```

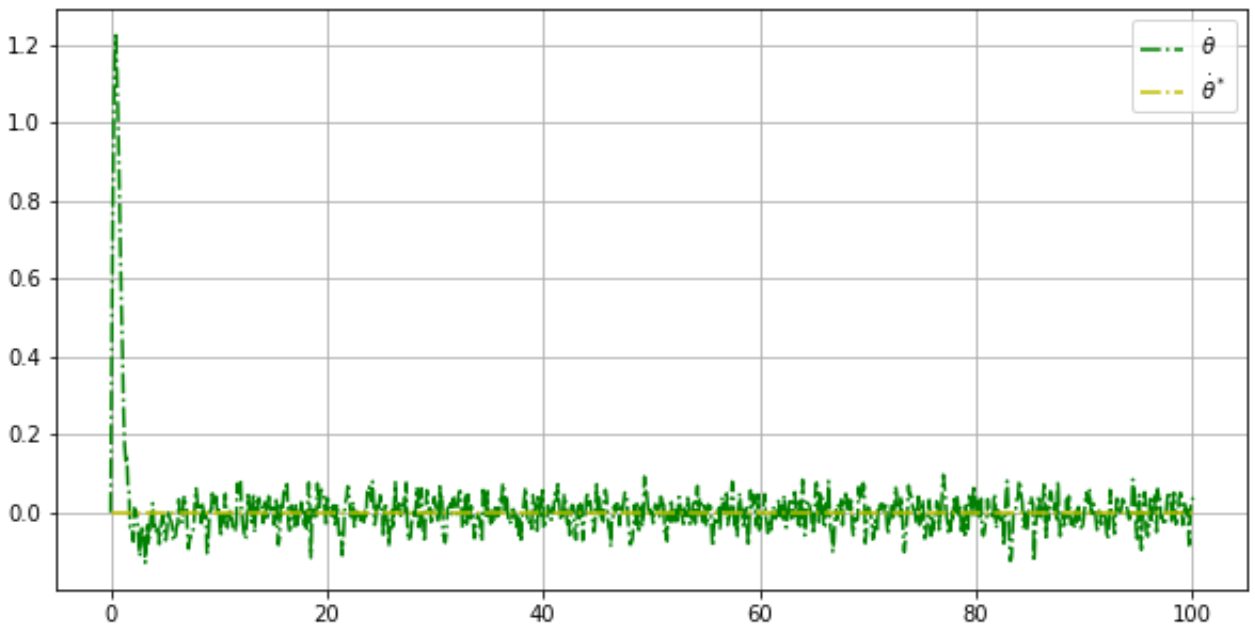


In [21]:

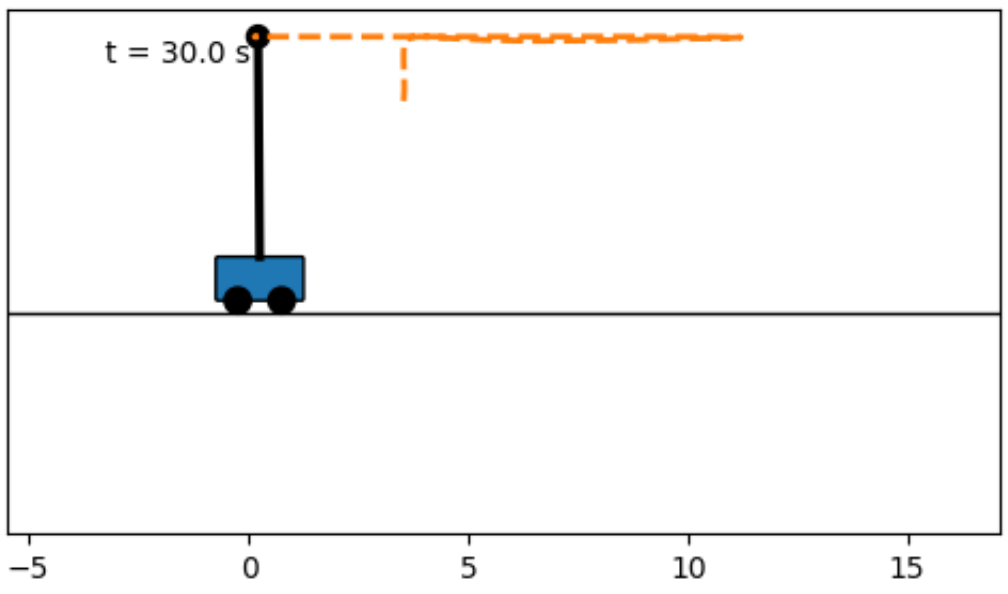
```
s_t = simulate_sys(add_noise=True, part_e = False, Q_multiply=1)

#Now plot the pos vs time
plt.figure(figsize=(10, 5))
plt.grid() #this allows us to see the grid
plt.plot(t,s_t[:,0],'r',label='$x$')
plt.plot(t, np.zeros(t.shape), 'y', label='$x^*$')
plt.legend(fontsize=10) #show the legend

plt.show() #this says display the info here


#Now plot the pos vs time
plt.figure(figsize=(10, 5))
plt.grid() #this allows us to see the grid
plt.plot(t,s_t[:,2],'-.r',label='$\\dot{x}$')
plt.plot(t, np.zeros(t.shape), '-.y', label='$\\dot{x}^*$')
plt.legend(fontsize=10) #show the legend

plt.show() #this says display the info here


#Now plot the angle vs time
plt.figure(figsize=(10, 5))
plt.grid() #this allows us to see the grid
plt.plot(t,s_t[:,1],'g',label='$\\theta$')
plt.plot(t,np.ones(t.shape)*np.pi,'y',label='$\\theta^*$')
plt.legend(fontsize=10) #show the legend

plt.show() #this says display the info here


#Now plot the angle vs time
plt.figure(figsize=(10, 5))
plt.grid() #this allows us to see the grid
plt.plot(t,s_t[:,3],'-.g',label='$\\dot{\\theta}$')
plt.plot(t,np.zeros(t.shape),'-.y',label='$\\dot{\\theta}^*$')
plt.legend(fontsize=10) #show the legend

plt.show() #this says display the info here
```



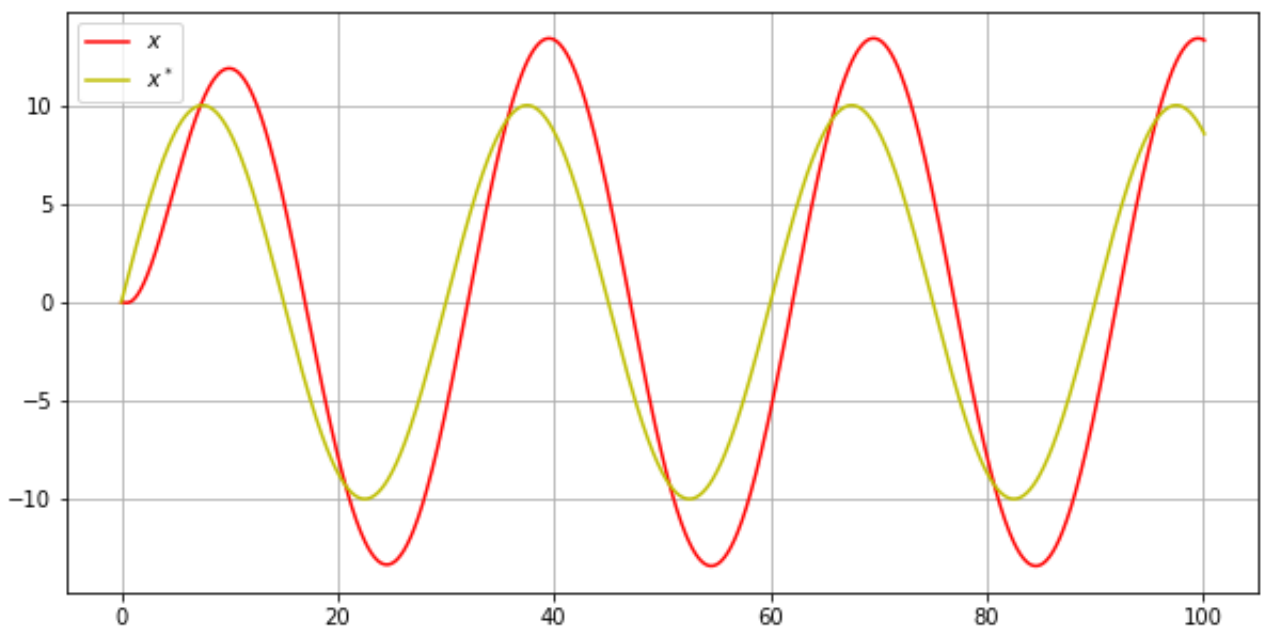


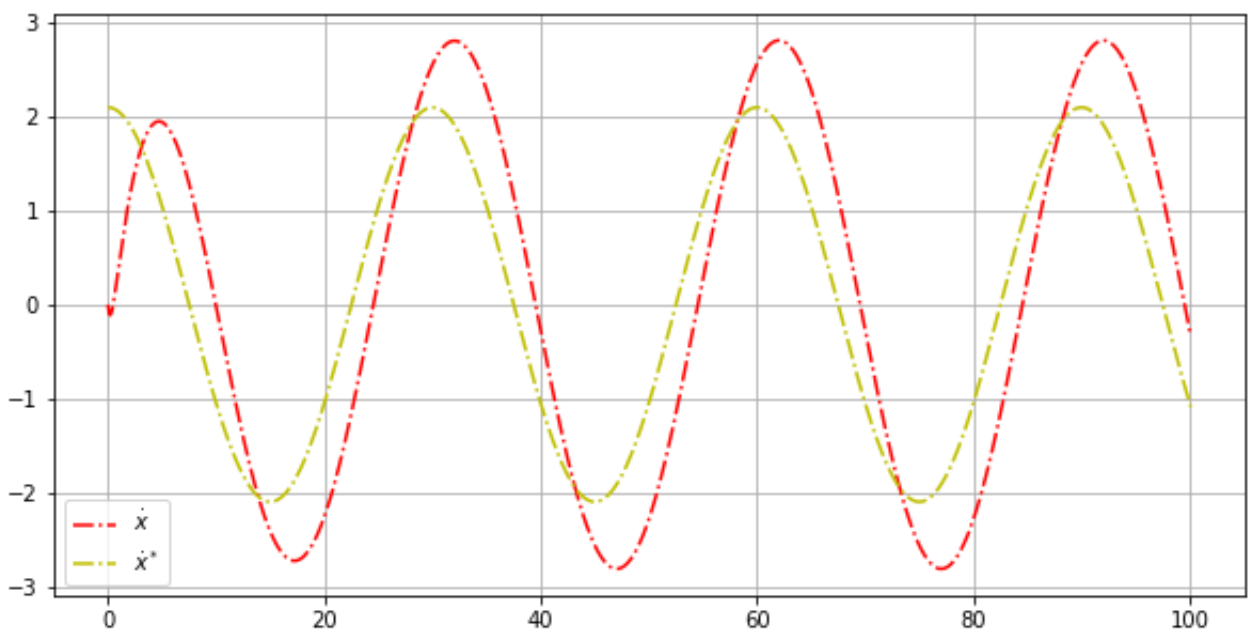


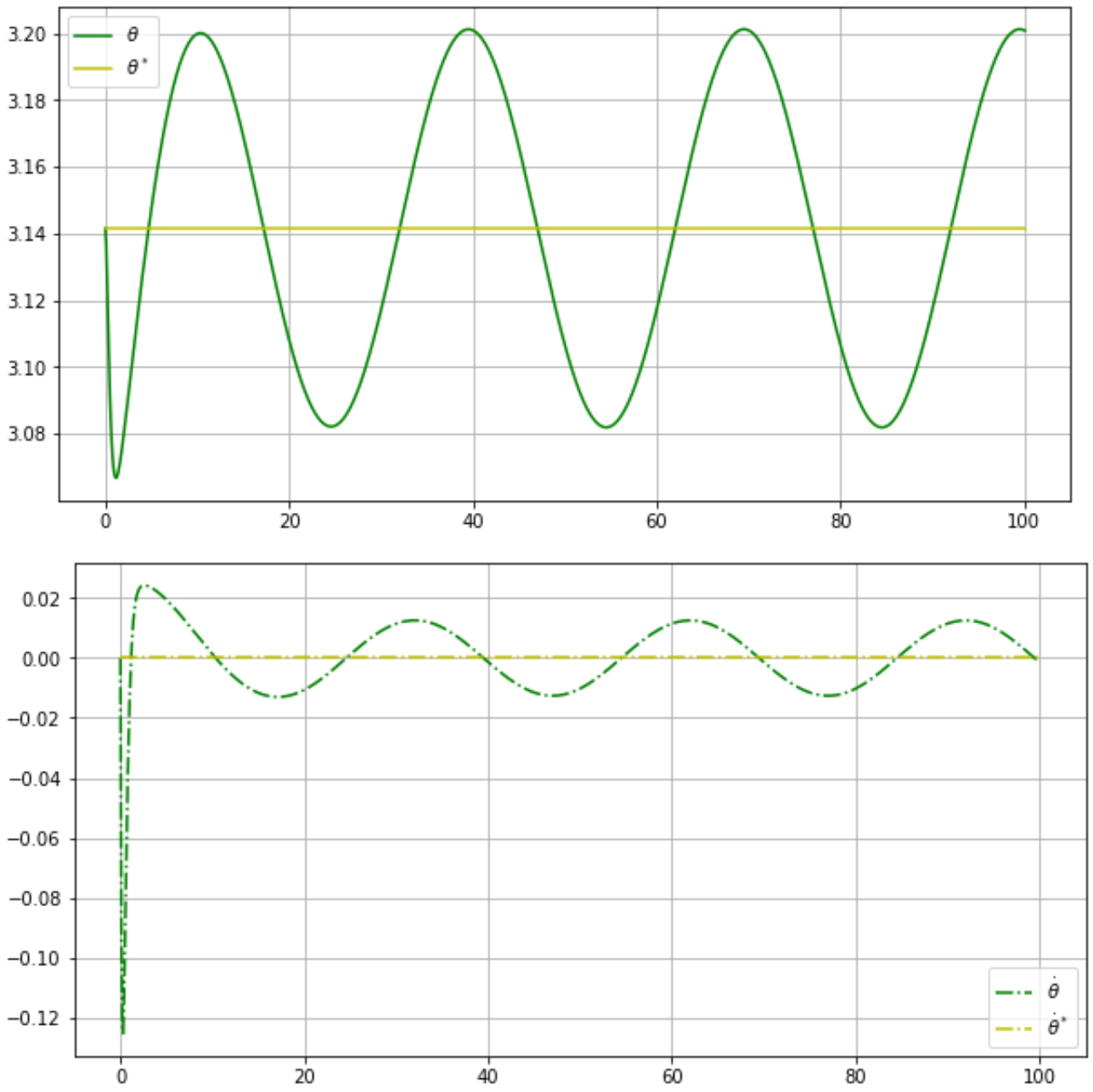
In [10]:
```
x = s_t[:,0]
θ = s_t[:,1]
fig, ani = animate_cartpole(t[:-1], x, θ)
ani.save('cartpole_d.mp4', writer='ffmpeg')
plt.show()
```



In [22]:
```
s_t = simulate_sys(add_noise=False, part_e = True, Q_multiply=1)

#Now plot the angle vs time
plt.figure(figsize=(10, 5))
plt.grid() #this allows us to see the grid
plt.plot(t,s_t[:,0],'r',label='$x$')
plt.plot(t, 10*np.sin(2*np.pi*t/30), 'y', label='$x^*$')
plt.legend(fontsize=10) #show the legend

plt.show() #this says display the info here


#Now plot the angle vs time
plt.figure(figsize=(10, 5))
plt.grid() #this allows us to see the grid
plt.plot(t,s_t[:,2],'-.r',label='$\\dot{x}$')
plt.plot(t, 10*(2*np.pi/30)*np.cos(2*np.pi*t/30), '-.y', label='$\\dot{x}^*$')
plt.legend(fontsize=10) #show the legend

plt.show() #this says display the info here


#Now plot the angle vs time
plt.figure(figsize=(10, 5))
plt.grid() #this allows us to see the grid
plt.plot(t,s_t[:,1],'g',label='$\\theta$')
plt.plot(t,np.ones(t.shape)*np.pi,'y',label='$\\theta^*$')
plt.legend(fontsize=10) #show the legend

plt.show() #this says display the info here


#Now plot the angle vs time
plt.figure(figsize=(10, 5))
plt.grid() #this allows us to see the grid
plt.plot(t,s_t[:,3],'-.g',label='$\\dot{\\theta}$')
plt.plot(t,np.zeros(t.shape),'-.y',label='$\\dot{\\theta}^*$')
plt.legend(fontsize=10) #show the legend

plt.show() #this says display the info here
```

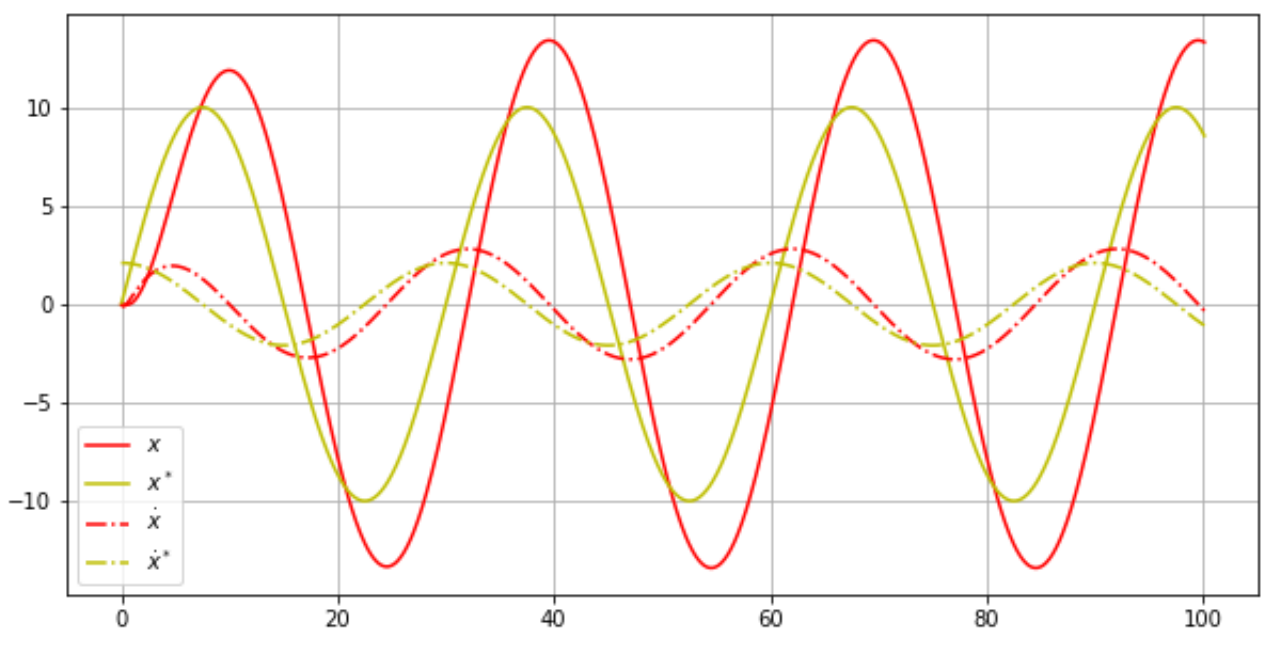


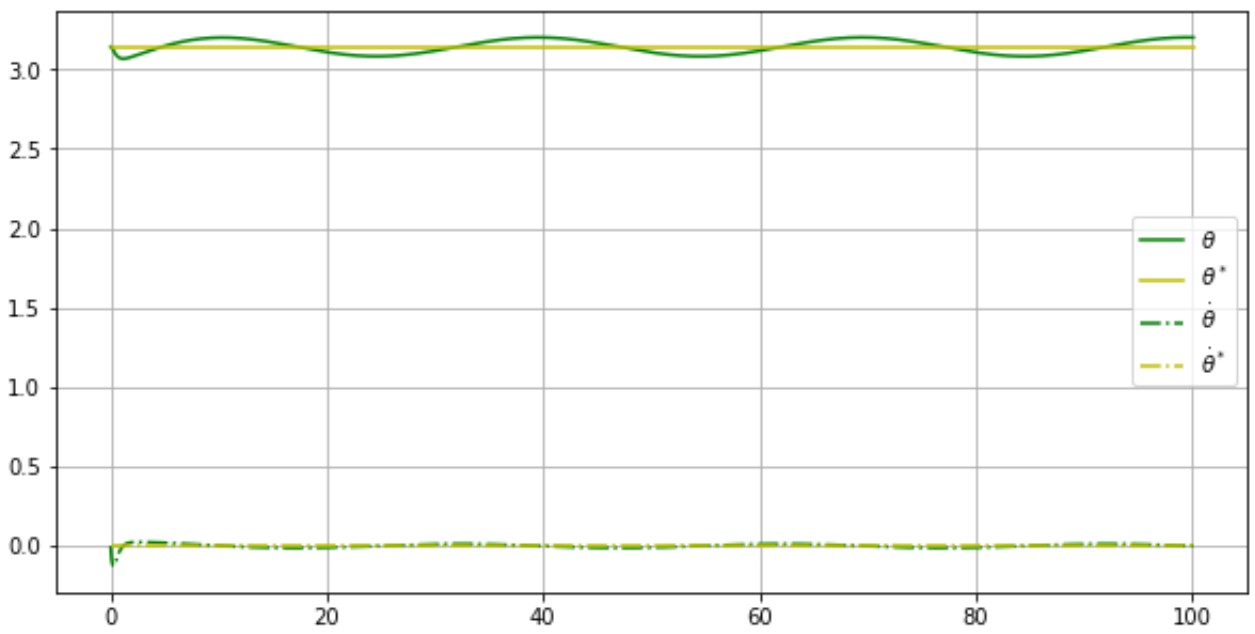




In [23]:

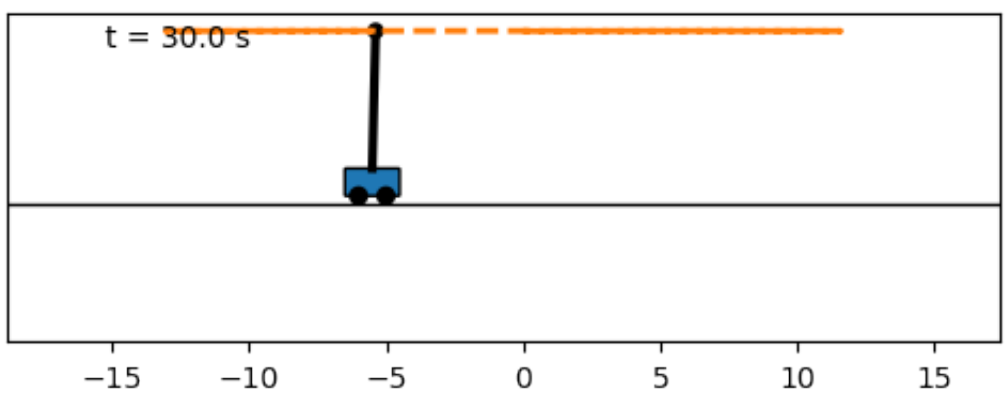
```
#Now plot the angle vs time
plt.figure(figsize=(10, 5))
plt.grid() #this allows us to see the grid
plt.plot(t,s_t[:,0],'r',label='$x$')
plt.plot(t, 10*np.sin(2*np.pi*t/30), 'y', label='$x^*$')
plt.plot(t,s_t[:,2],'-.r',label='$\\dot{x}$')
plt.plot(t, 10*(2*np.pi/30)*np.cos(2*np.pi*t/30), '-.y', label='$\\dot{x}^*$')
plt.legend(fontsize=10) #show the legend

plt.show() #this says display the info here


#Now plot the angle vs time
plt.figure(figsize=(10, 5))
plt.grid() #this allows us to see the grid
plt.plot(t,s_t[:,1],'g',label='$\\theta$')
plt.plot(t,np.ones(t.shape)*np.pi,'y',label='$\\theta^*$')
plt.plot(t,s_t[:,3],'-.g',label='$\\dot{\\theta}$')
plt.plot(t,np.zeros(t.shape),'-.y',label='$\\dot{\\theta}^*$')
plt.legend(fontsize=10) #show the legend

plt.show() #this says display the info here
```





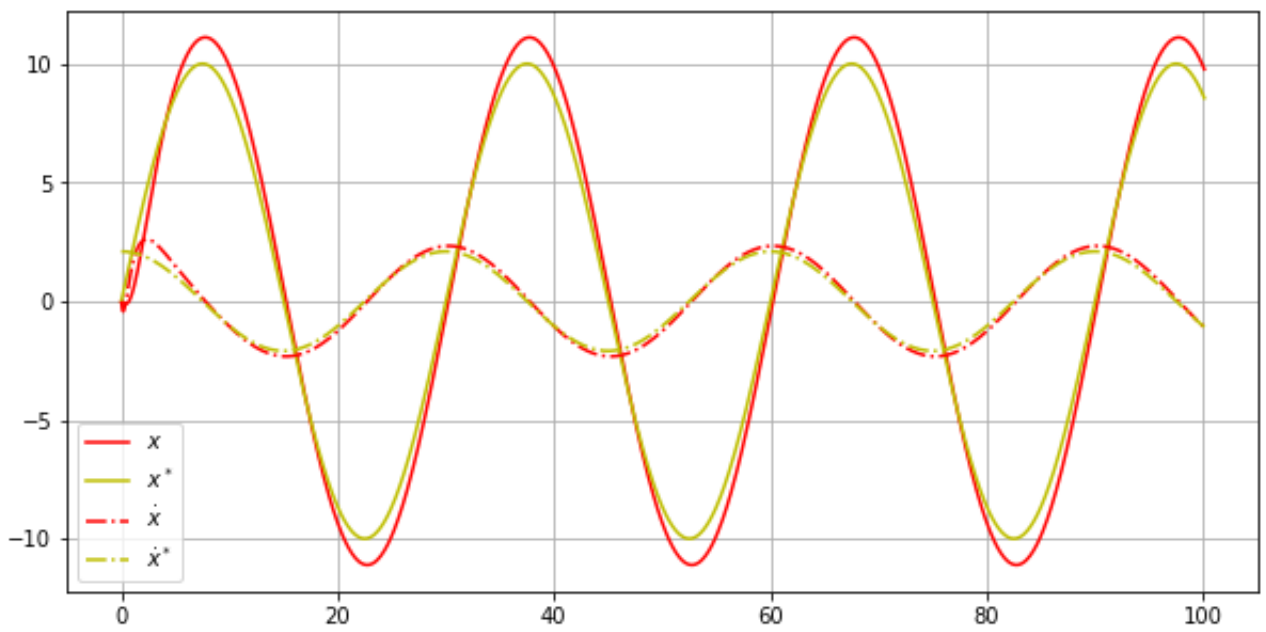
In [13]:

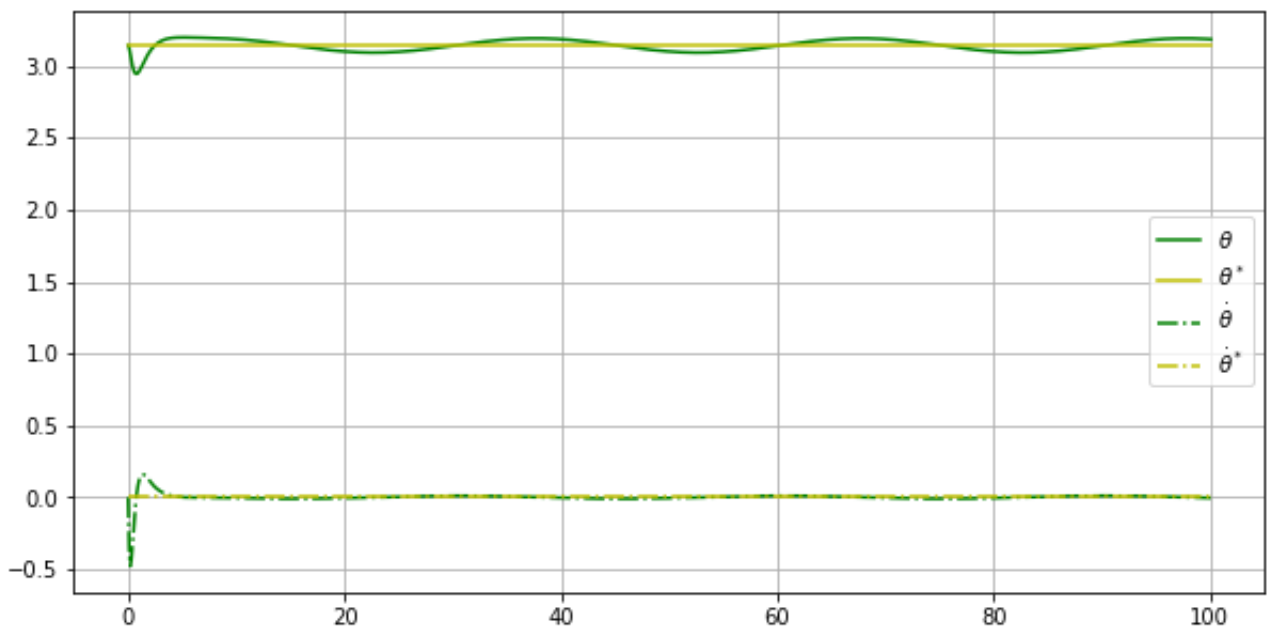
```
x = s_t[:,0]
θ = s_t[:,1]
fig, ani = animate_cartpole(t[:-1], x, θ)
ani.save('cartpole_e.mp4', writer='ffmpeg')
plt.show()
```



In [24]:

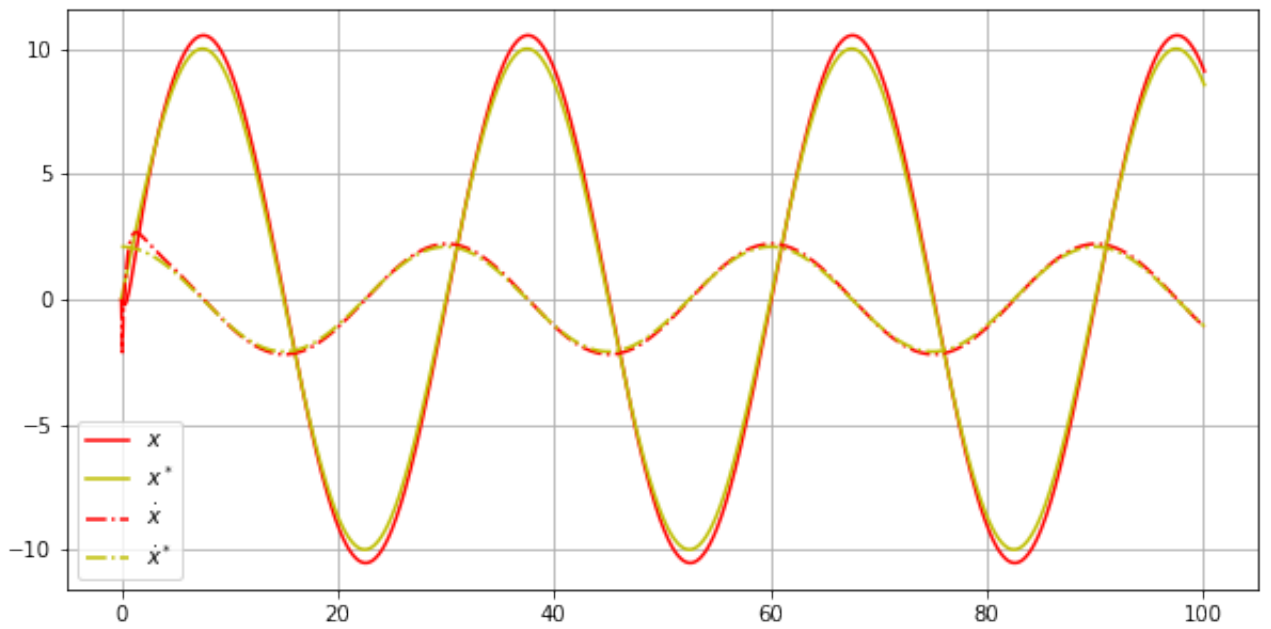
```
s_t = simulate_sys(add_noise=False, part_e = True, Q_multiply=100)

#Now plot the angle vs time
plt.figure(figsize=(10, 5))
plt.grid() #this allows us to see the grid
plt.plot(t,s_t[:,0],'r',label='$x$')
plt.plot(t, 10*np.sin(2*np.pi*t/30), 'y', label='$x^*$')
plt.plot(t,s_t[:,2],'-.r',label='$\\dot{x}$')
plt.plot(t, 10*(2*np.pi/30)*np.cos(2*np.pi*t/30), '-.y', label='$\\dot{x}^*$')
plt.legend(fontsize=10) #show the legend

plt.show() #this says display the info here


#Now plot the angle vs time
plt.figure(figsize=(10, 5))
plt.grid() #this allows us to see the grid
plt.plot(t,s_t[:,1],'g',label='$\\theta$')
plt.plot(t,np.ones(t.shape)*np.pi,'y',label='$\\theta^*$')
plt.plot(t,s_t[:,3],'-.g',label='$\\dot{\\theta}$')
plt.plot(t,np.zeros(t.shape),'-.y',label='$\\dot{\\theta}^*$')
plt.legend(fontsize=10) #show the legend

plt.show() #this says display the info here
```

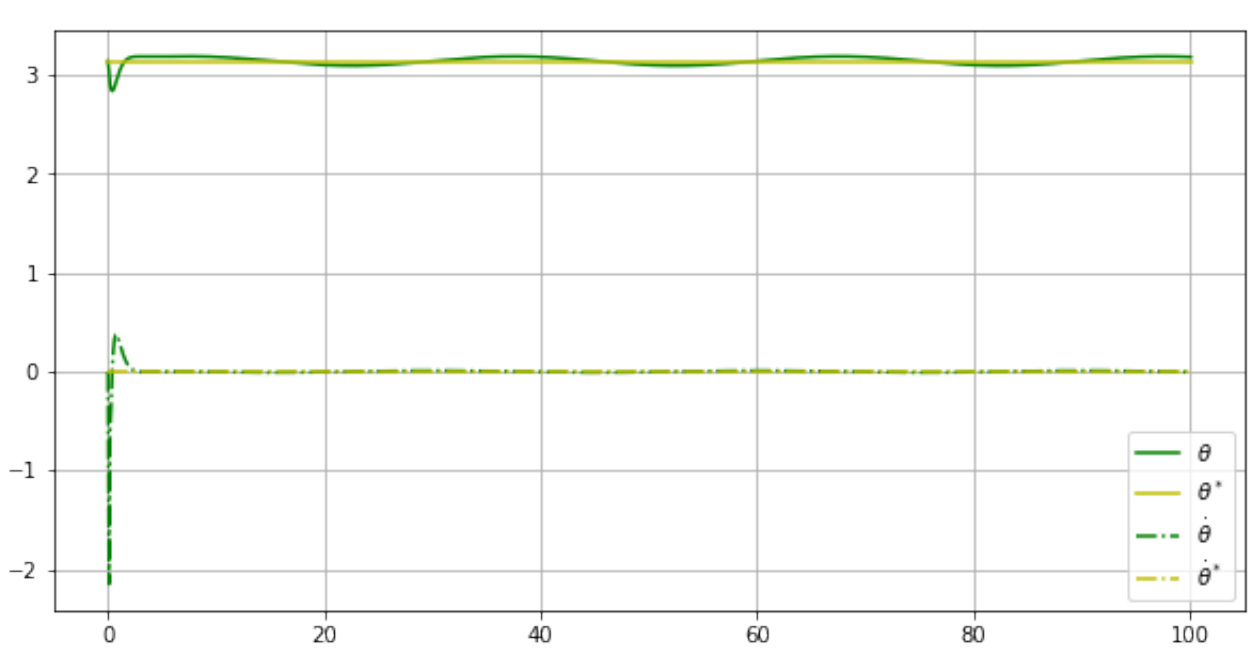




In [27]:

```
s_t = simulate_sys(add_noise=False, part_e = True, Q_multiply=100000)

#Now plot the angle vs time
plt.figure(figsize=(10, 5))
plt.grid() #this allows us to see the grid
plt.plot(t,s_t[:,0],'r',label='$x$')
plt.plot(t, 10*np.sin(2*np.pi*t/30), 'y', label='$x^*$')
plt.plot(t,s_t[:,2],'-.r',label='$\\dot{x}$')
plt.plot(t, 10*(2*np.pi/30)*np.cos(2*np.pi*t/30), '-.y', label='$\\dot{x}^*$')
plt.legend(fontsize=10) #show the legend

plt.show() #this says display the info here


#Now plot the angle vs time
plt.figure(figsize=(10, 5))
plt.grid() #this allows us to see the grid
plt.plot(t,s_t[:,1],'g',label='$\\theta$')
plt.plot(t,np.ones(t.shape)*np.pi,'y',label='$\\theta^*$')
plt.plot(t,s_t[:,3],'-.g',label='$\\dot{\\theta}$')
plt.plot(t,np.zeros(t.shape),'-.y',label='$\\dot{\\theta}^*$')
plt.legend(fontsize=10) #show the legend

plt.show() #this says display the info here
```





In [ ]: