HW3

March 30, 2020

[0]:
```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt

np.random.seed(42)

time = np.linspace(0, 1, 1000)    # interval from 0 to 1
```

1 Proportional-derivative (PD) control

Consider a second order linear ODE:

$$\ddot{x} + \mu\dot{x} + kx = u$$

We can use P control:

$$u = k_p(x^* - x)$$

or we can add a derivative term:

$$u = k_d(\dot{x}^* - \dot{x}) + k_p(x^* - x)$$

This is a **PD controller**.
Let us introduce $e = x^* - x$. Then we have:

$$u = k_d\dot{e} + k_p e$$

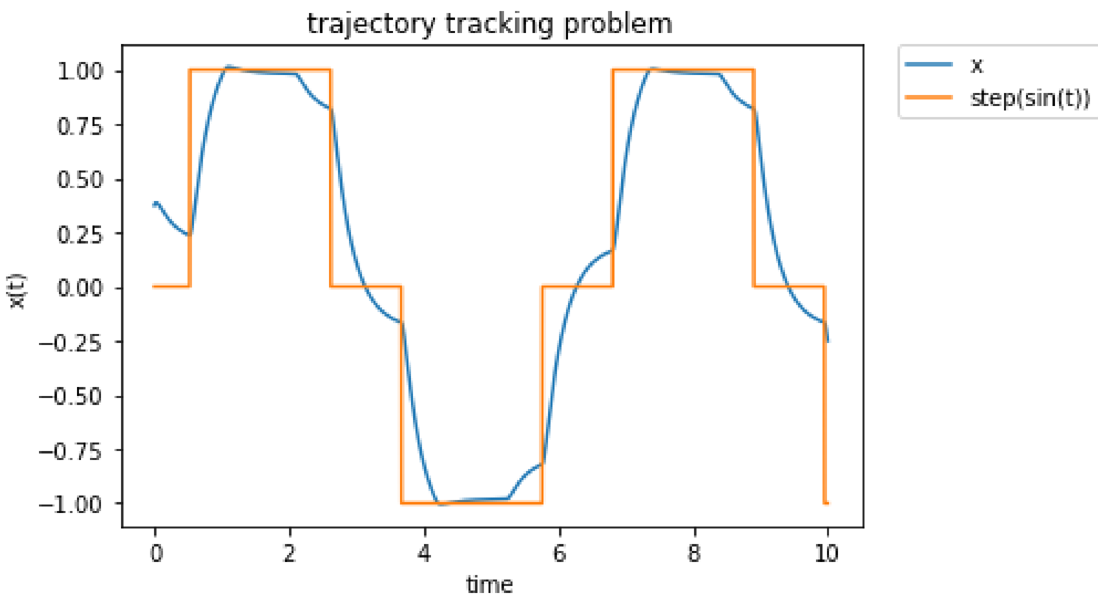
Variable e here is a **control error**.

2 Part A

Design PD-controller that tracks time varying reference states i.e. [x(t),x (t)] as closely as possible. Test your controller on different trajectories, at least two. System: x + x + kx = u, see variants below. ### Trajectory 1: [step(sin(t)), step(cos(t))] ### Trajectory 2: [step(cos(t)), step(sin(t))]

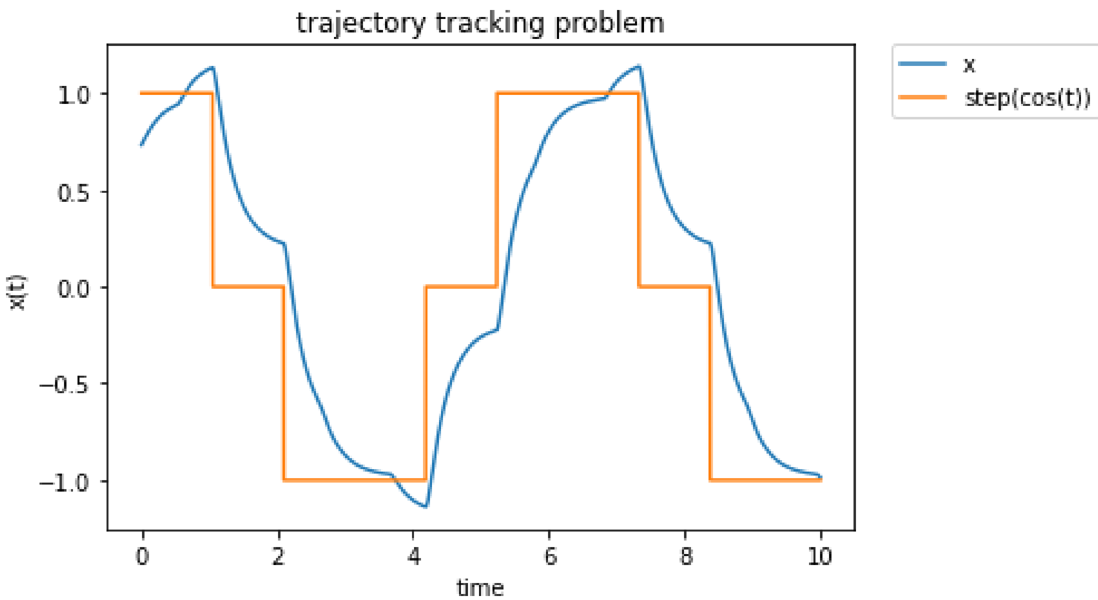
[2]:
```
mu = 13
k = 2

t_steps=10000
```

```python
kp = 100
kd = 20


def trajectory(t):
  return np.round(np.sin(t))

def trajectory_dot(t):
  return np.round(np.cos(t))

def PD(x,t):
  x_desired = trajectory(t)
  x_dot_desired = trajectory_dot(t)
  error     = x_desired     - x[0]
  error_dot = x_dot_desired - x[1]
  u = kp*error + kd*error_dot
  return np.array([x[1], (u - mu*x[1] - k*x[0])])

x0 = np.random.rand(2)
time = np.linspace(0, 10, t_steps)

sol=odeint(PD, x0, time)
plt.plot(time, sol[:,0],label="x")
plt.plot(time, trajectory(time),label="step(sin(t))")
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
plt.xlabel('time')
plt.ylabel('x(t)')
plt.title("trajectory tracking problem")
plt.show()
```



[3]:
```
mu = 13
k = 2

t_steps=10000

kp = 100
kd = 20


def trajectory(t):
  return np.round(np.cos(t))

def trajectory_dot(t):
  return np.round(np.sin(t))

def PD(x,t):
  x_desired = trajectory(t)
  x_dot_desired = trajectory_dot(t)
  error     = x_desired     - x[0]
  error_dot = x_dot_desired - x[1]
  u = kp*error + kd*error_dot
  return np.array([x[1], (u - mu*x[1] - k*x[0])])

x0 = np.random.rand(2)
time = np.linspace(0, 10, t_steps)

sol=odeint(PD, x0, time)
```

```
plt.plot(time, sol[:,0],label="x")
plt.plot(time, trajectory(time),label="step(cos(t))")
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
plt.xlabel('time')
plt.ylabel('x(t)')
plt.title("trajectory tracking problem")
plt.show()
```



2.1 Part B and C

Tune controller gains kp and kd . Find gains that provide no ocsillations and no overshoot. Prove it with step input. Prove that controlled oscillator dynamics is stable for your choice of kp and kd.

[4]:
```
A=np.array([[-mu,-k],[1,0]])
B=np.array([[1],[0]])
ks=np.array([[kd,kp]])
np.linalg.eigvals(A-B.dot(ks))
```

[4]:
```
array([-29.54798835,  -3.45201165])
```

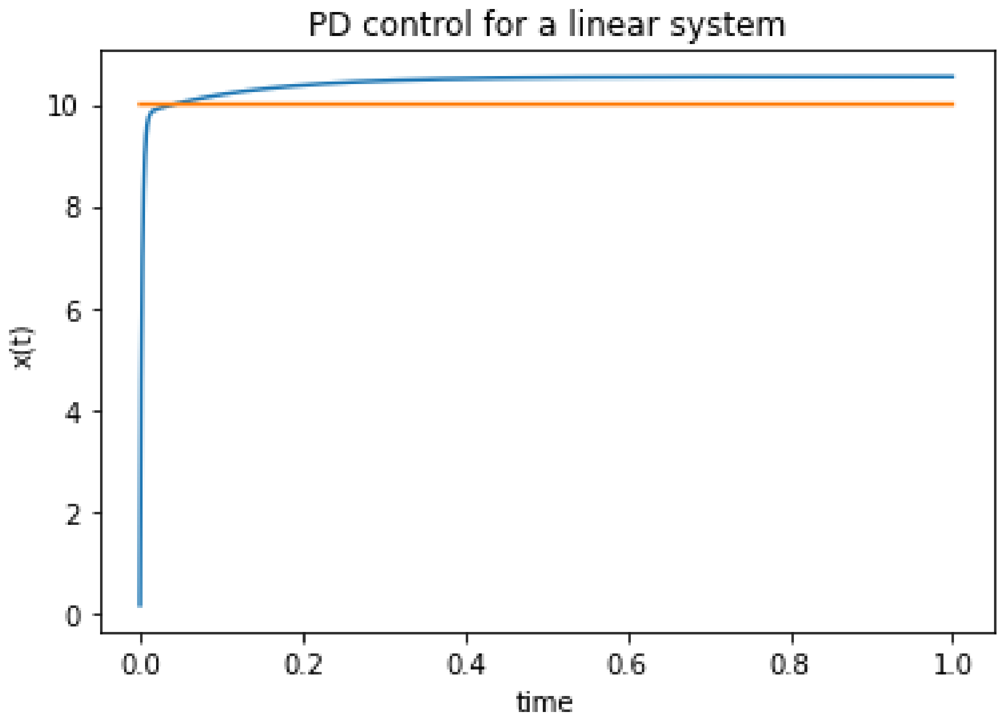
As we see Im(eigenvalues) = 0, that means we will have no **ocsillations**
And because Re(eigenvalues) < 0 for all eigenvalues this system is **stable**

2.2 Part D

2.2.1 Think of how you would implement PD control for a linear system:

[5]:
```
A = np.array([[10, 3], [5, -5]])
B=np.array([[1],[1]])
k=np.array([[400,20]])
print(np.linalg.eigvals(A-B.dot(k)))

t_steps=10000
time = np.linspace(0, 1, t_steps)
x0=np.random.rand(2)

def trajectory(t):
  return t*0+10

def trajectory_dot(t):
  return 0

def PD(x,t):
  x_desired = np.array([trajectory(t) , trajectory_dot(t)])
  error     = np.array([x_desired[0] - x[0] , x_desired[1] - x[1]])

  u=k.dot(error)
  return A.dot(x)+B.dot(u)


sol=odeint(PD, x0, time)


plt.plot(time, sol[:,0])
plt.xlabel('time')
plt.ylabel('x(t)')
plt.plot(time, trajectory(time),label="trajectory")
plt.title('PD control for a linear system')
plt.show()
```

```
[-407.55311795   -7.44688205]
```



3 Part E

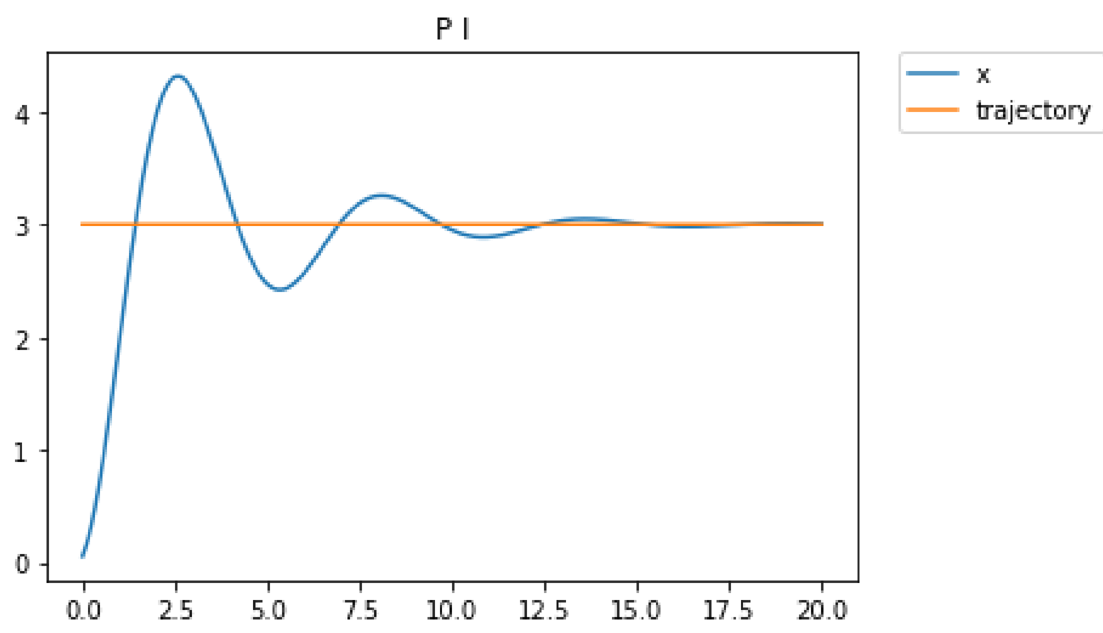
3.1 Implement a PI/PID controller for the system: x + x + kx + 9.8 = u Test your controller on different trajectories, at least two.

3.1.1 Trajectory 1: PI with x_desired = 3

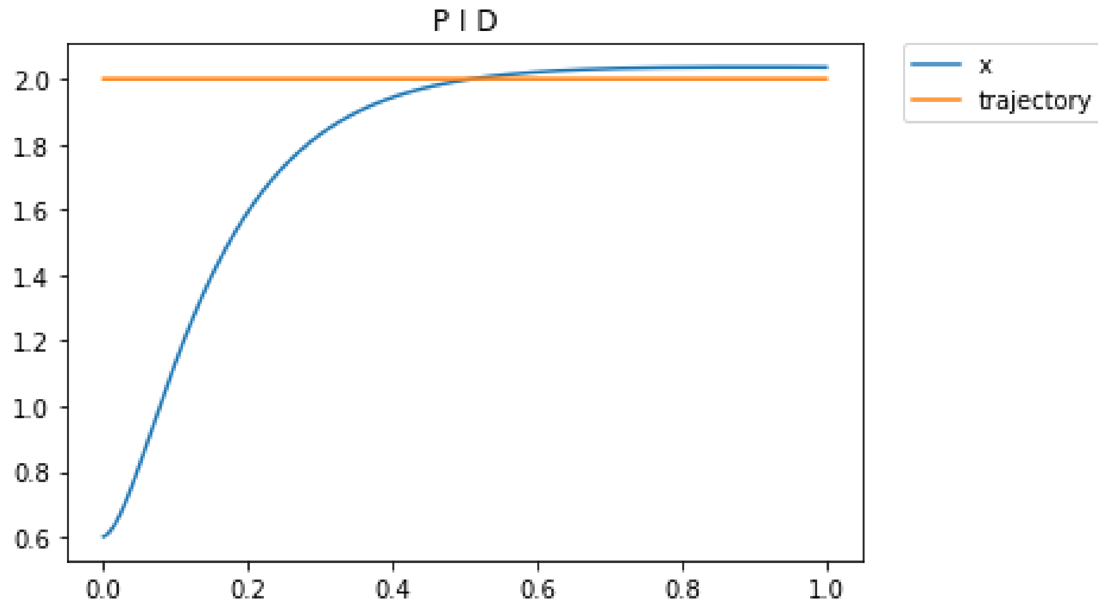
3.1.2 Trajectory 2: PID with x_desired = 2

[6]:
```
mu = 13
k = 2

kp=8
kd=2
ki=20

x=np.random.rand(2)

t_steps=10000
begin=0
end=20
step=(end-begin)/t_steps
graph=[]
```

```python
def trajectory(t):
  return t*0+3

def trajectory_dot(t):
  return (trajectory(t+step) - trajectory(t))/step


sum=0

time=np.linspace(begin,end,t_steps)


for t in time:
  graph.append(x[0])
  x_desired = trajectory(t)
  x_dot_desired = trajectory_dot(t)
  error     = x_desired     - x[0]
  error_dot = x_dot_desired - x[1]
  u = kp*error + kd*error_dot+ki*sum
  sum+=(error)*step
  x[0],x[1] = x[0]+x[1]*step,  x[1]+(u - mu*x[1] - k*x[0] - 9.8)*step

plt.plot(time, graph,label="x")
plt.plot(time, trajectory(time),label="trajectory")
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
plt.title("P I")
plt.show()
```



[7]:
```
mu = 13
k = 2
kp=200
kd=20
ki=100

x=np.random.rand(2)


t_steps=10000
begin=0
end=1
step=(end-begin)/t_steps
graph=[]

sum=0


time=np.linspace(begin,end,t_steps)

def trajectory(t):
  return t*0+2

def trajectory_dot(t):
  return (trajectory(t+step) - trajectory(t))/step


for t in time:
  graph.append(x[0])
  x_desired = trajectory(t)
  x_dot_desired = trajectory_dot(t)
  error     = x_desired     - x[0]
  error_dot = x_dot_desired - x[1]
  u = kp*error + kd*error_dot+ki*sum
  sum+=error*step
  x[0],x[1] = x[0]+x[1]*step, x[1]+(u - mu*x[1] - k*x[0] - 9.8)*step

plt.plot(time, graph,label="x")
plt.plot(time, trajectory(time),label="trajectory")
plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left', borderaxespad=0.)
plt.title("P I D")
plt.show()
```


P I D
x
trajectory
2.0
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.0
0.2
0.4
0.6
0.8
1.0