Questions:

1. Describe qualitatively what you observe when you increase/ decrease P and D.

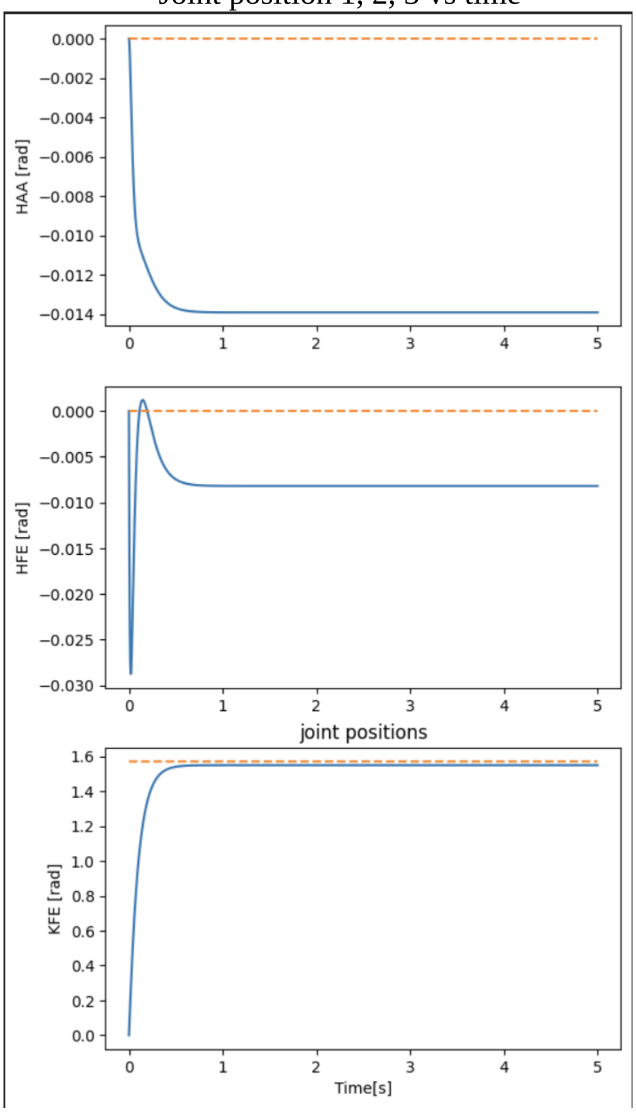
   When increasing P and D, the robot follows the trajectory and moves to the destinate position in a much steadier manner, with less shaking on all joints. This is reflected in the graph plotted at the end, that with larger P and D, there will be fewer oscillations.

2. Tune the P and D gains to have a good tracking of the positions $[0,0,\frac{\pi}{2}]$ without any oscillations. The P and D gains need not be the same for different joints. What gains did you find? Plot the position and velocities of each joint as a function of time with these gains. (starting from the original initial robot configuration).

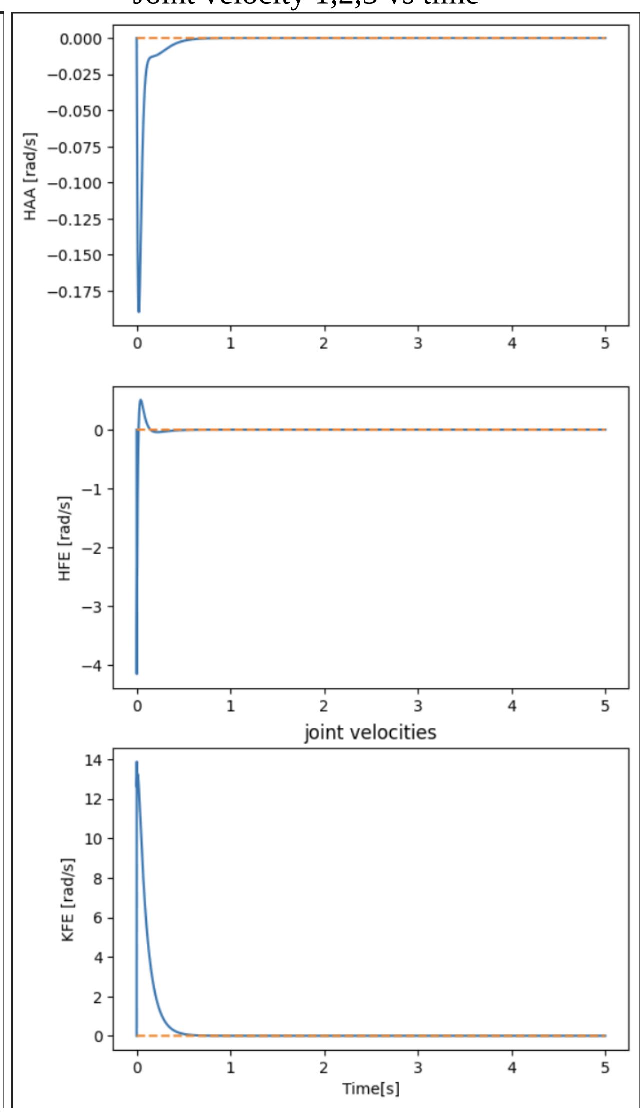
   P and D were modified to be [5,5,2] and [0.5,0.5,0.2] respectively so oscillations can be reduced to almost 0.

```
# the PD gains - they are constant so we define them outside the control loop
P = np.array([5, 5, 2])
D = np.array([0.5, 0.5, 0.2])
```

Joint position 1, 2, 3 vs time



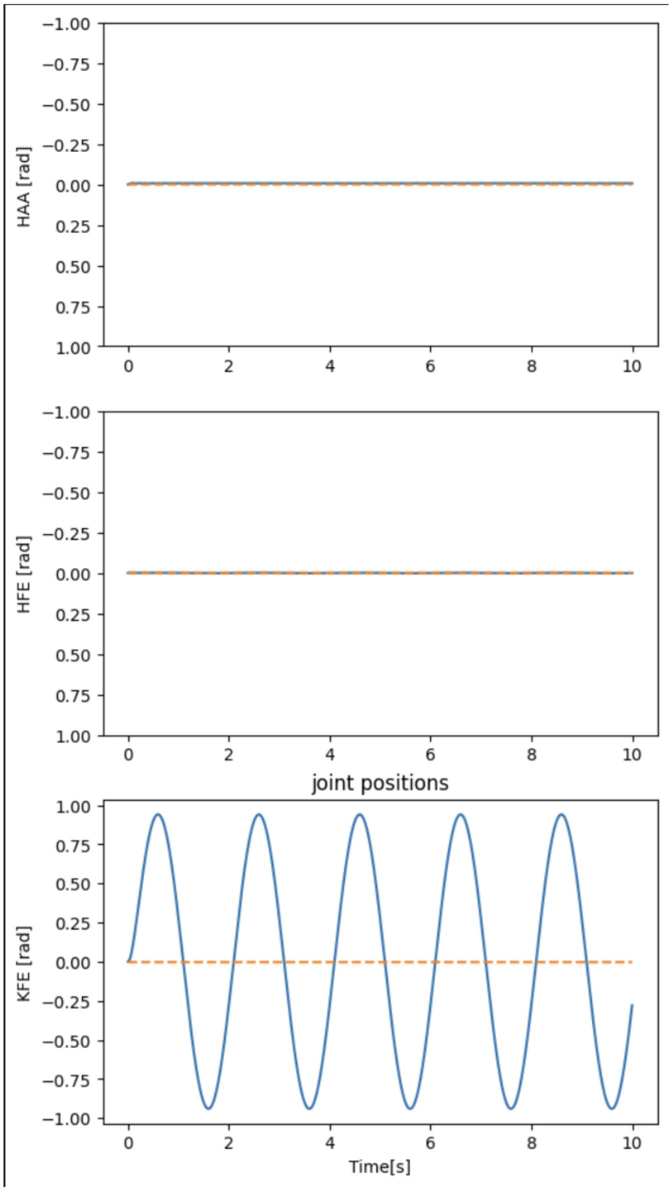
Joint velocity 1,2,3 vs time



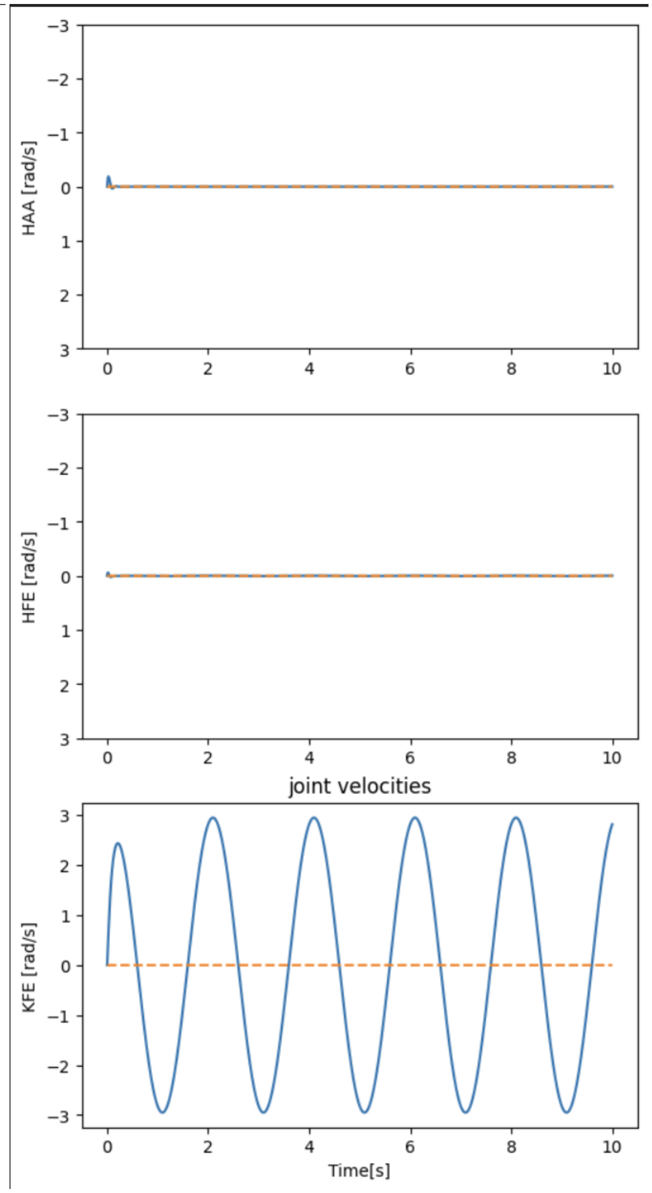
3. Use the PD controller to do the following task: keep the position of the first two joints fixed and follows the following position trajectory for the last joint $0.8\sin(\pi t)$. Plot the results (positions and velocities as a function of time for all joints). Simulate for at least 10 seconds.

```
for i in range(num_steps):
    # get the current time
    time[i] = dt * i
    # we read the position and velocities of the joints from the robot or simulation
    q, dq = robot.get_state()

    # we store these values for later use
    measured_positions[i,:] = q
    measured_velocities[i,:] = dq

    q_des = np.array([0.,0., np.sin(np.pi*time[i])])

    error = q_des - q # the position error for all the joints (it's a 3D vector)
    d_error = dq_des-dq # the velocity error for all the joints

    # we compute the desired torques as a PD controller
    joint_torques = P * error + D * d_error
    desired_torques[i,:] = joint_torques

    # we send them to the robot and do one simulation step
    robot.send_joint_torque(joint_torques)
    robot.step()
```

Joint position 1, 2, 3 vs time



Joint velocity 1,2,3 vs time



4. Change the joint trajectories to get robot to draw a circle in the air with its fingertip.

   Theoretically speaking, since the position of the fingertip is a sinusoidal wave, to make it draw a circle, one can just change the sign of its position vector once it reaches the top. This could be achieved by adding a variable $prev\_q\_des$ that keeps tracking the change in q_des. If q_des starts to decrease, meaning it is changing direction, a minus sign can be assigned to make sure it keeps going in the same direction.

```
prev_q_des = [0.,0.,0.]
for i in range(num_steps):
    # get the current time
    time[i] = dt * i
    # we read the position and velocities of the joints from the robot or simulation
    q, dq = robot.get_state()

    # we store these values for later use
    measured_positions[i,:] = q
    measured_velocities[i,:] = dq
    ###
    q_real_des = np.array([0.,0., 3.5*np.sin(np.pi*time[i])])
    if q_real_des[2] < prev_q_des[2]:
        q_des[2] = - q_real_des[2]
    else:
        q_des[2] = q_real_des[2]

    #q_des = np.array([0.,0., 3*np.sin(np.pi*time[i])])
    error = q_des - q # the position error for all the joints (it's a 3D vector)
    if error[2] < 0:
        error[2] = -error[2]
    d_error = dq_des-dq # the velocity error for all the joints

    # we compute the desired torques as a PD controller
    joint_torques = P * error + D * d_error
    desired_torques[i,:] = joint_torques

    prev_q_des = q_real_des

    # we send them to the robot and do one simulation step
    robot.send_joint_torque(joint_torques)
    robot.step()
```

However, it seems that due to some reason, the tip won't go through the top point, as a result it will only stay there.