1 Task 1.

My name is *Anton Brisilin*, and my email is *a.brisilin@innopolis.university*. My generated variant is **F**.

2 Task 2.

Part A.

Design PD-controller that tracks time varying reference states, i.e. $[x^*(t), \dot{x}^*(t)]$ as closely as possible. Test your controller on different trajectories, at least two. System: $x + \mu x + kx = u$, see variants below.
For variant F, $\mu = 63, k = 15$ Our open-loop system equation is

$$\ddot{x} + 63\dot{x} + 15x = u,$$

where $x = x(t)$, $u = u(t)$
Our controller is required to be PD-controller. Hence, we have following formula for our input $u(t)$:

$$u(t) = K_p e(t) + K_d \dot{e}(t)$$

As I will use `odeint()` function from `scipy.integrate` module, I will need a formula for a derivative vector of $\vec{x} = [x(t), \dot{x}(t)]^T$: $\dot{\vec{x}} = [\dot{x}(t), \ddot{x}(t)]^T$. Hence I should convert my system to state-space form.

$$\begin{bmatrix} \dot{x} \\ \ddot{x} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -15 & -63 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u$$

Adding our controller, we get

$$\begin{bmatrix} \dot{x} \\ \ddot{x} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -15 & -63 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \begin{bmatrix} K_p & K_d \end{bmatrix} \left(\begin{bmatrix} x^* \\ \dot{x^*} \end{bmatrix} - \begin{bmatrix} x \\ \dot{x} \end{bmatrix} \right)$$

Now my goal is implement such a system in Python, you can find it in `Task2/pd.py`. First of all, I declared μ, k, right time boundary and step for numerical solution as constants

```
# constants
step = 0.001
final_time = 1
mu = 63
k = 15
```

Then I declared reference functions:

```
import numpy as np
import matplotlib.pylab as plt
from scipy.integrate import odeint

# constants
step = 0.001
final_time = 1
mu = 63
k = 15

# function to obtain reference plot from function
def get_ref(count,step,func):
    res = []
    for i in np.arange(0,count,step):
        res.append(func(i))
    return [np.linspace(0,count,int(count/step)),res]

# desired functions
def f_des(t):
    return 1
def f_dot_des(t):
    return 0
```

Then, I created a function that will compute our derivative vector:

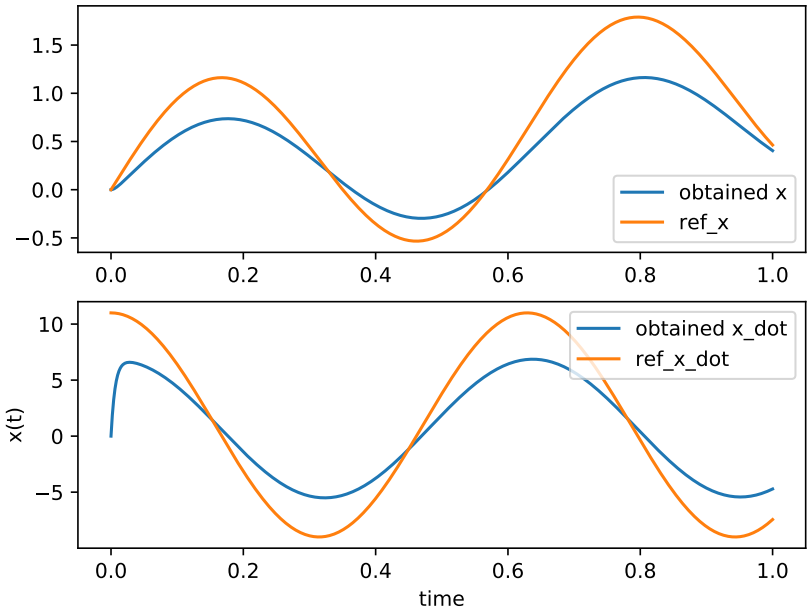
```
def controlled_system(x,t):
    error = f_des(t) - x[0]
    error_dot = f_dot_des(t) - x[1]
    u = kp*error + kd*error_dot
    return np.array([x[1],(u - mu*x[1] - k*x[0])])
```

Using these functions, I can simulate work of the system, and try to tune the coefficients:

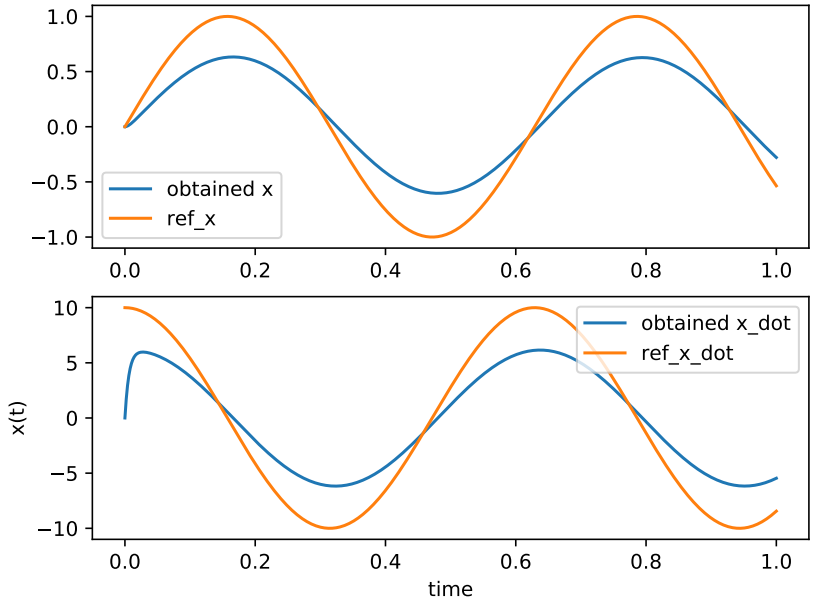
```
times = np.linspace(0,final_time,int(final_time/step))
solution = odeint(controlled_system,[0,0],times)

plt.subplot(2, 1, 1)
plt.plot(times, solution[:,0], label = 'obtained x')
plt.plot(times, ref_x, label = 'ref_x')
plt.legend()
plt.subplot(2, 1, 2)
plt.plot(times, solution[:,1], label = 'obtained x_dot')
plt.plot(times, ref_x_dot, label = 'ref_x_dot')
plt.xlabel('time')
plt.ylabel('x(t)')
plt.legend()
plt.show()
```

Testing on different trajectories

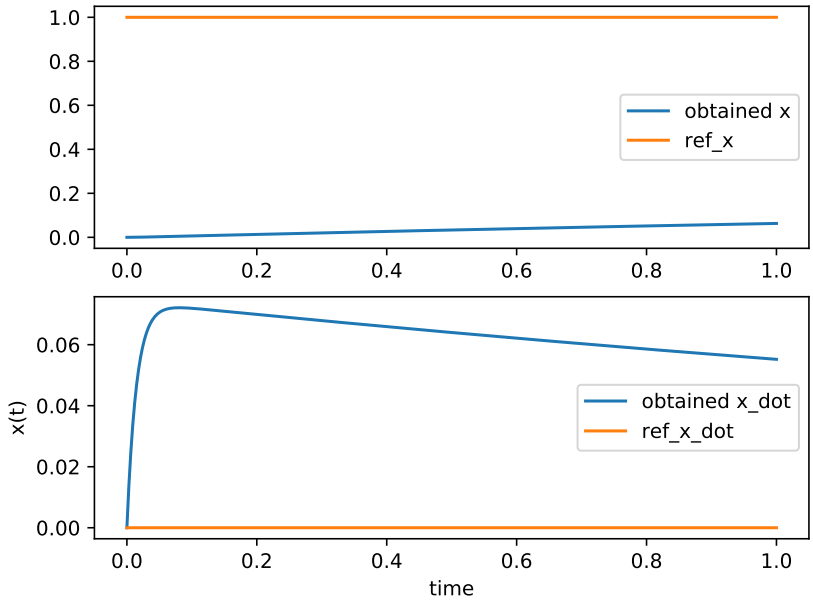
As a first reference trajectory I have chosen $x^*(t) = sin(10t) + t$, $\dot{x}^*(t) = 10cos(10t) + 1$, with zero initial conditions. Here are the plots of solution, obtained with $K_p = 100$, $K_d = 100$.



As a second reference trajectory I have chosen $x^*(t) = sin(10t)$, $\dot{x}^*(t) = 10cos(10t)$, with zero initial conditions. Here are the plots of solution, obtained with $K_p = 100$, $K_d = 100$.



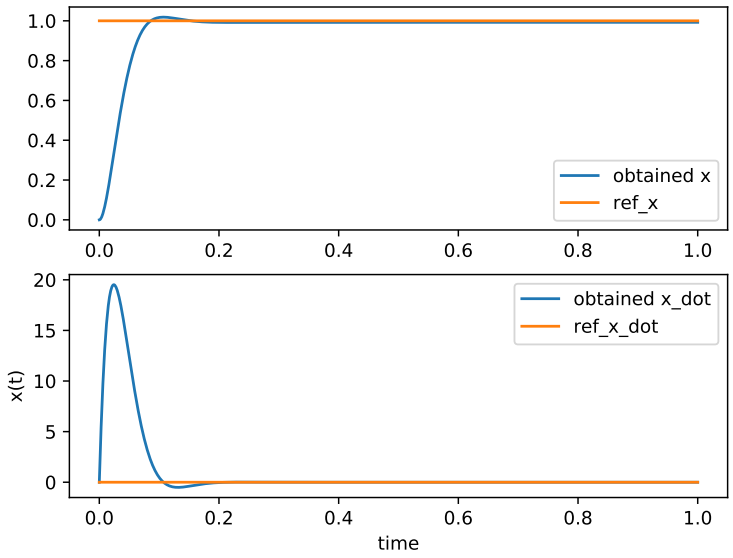
Third reference trajectory is just a step function $x^*(t) = 1$, $\dot{x}^*(t) = 0$, with zero initial conditions. Here are the plots of solution, obtained with $K_p = 5$, $K_d = 5$.



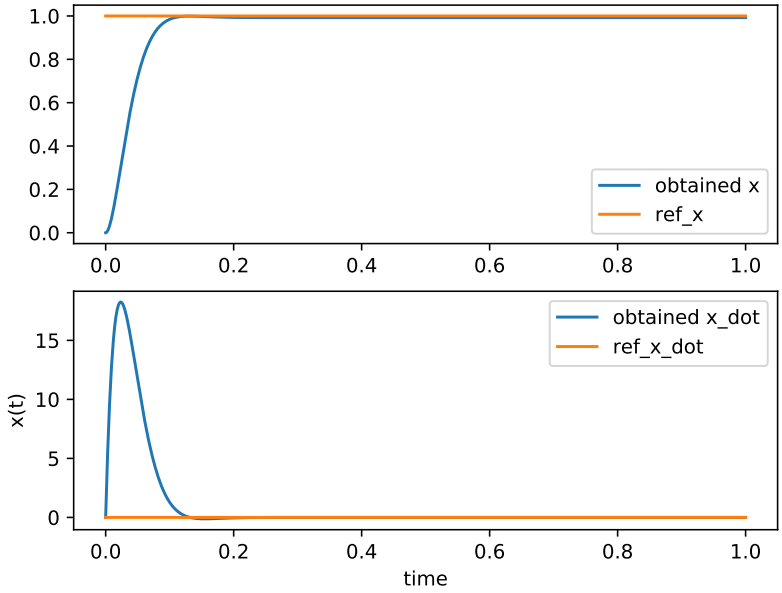
As one can see, my controller poorly follows desired trajectory and need to be tuned for each trajectory individually.

Part B. Controller tuning

We begin with $K_p = 5$, $K_d = 5$. As you can see on the above plot, it gives us a large steady-state error, so K_p should be increased.
I had been increasing it, until I got steady-state error $e \approx 10^{-2}$. The controller gains are now $K_p = 2000$, $K_d = 5$



Now we have almost no steady-state error, but there is an overshoot appeared now. Therefore, K_d should be increased.
I had been increasing it, until I got rid of overshoot. The controller gains now are $K_p = 2000$, $K_d = 13$



Now there is no overshoot, and steady-state error $e \approx 10^{-2}$, as it can be seen from the plot. The gains are $K_p = 2000$, $K_d = 13$.

Part C. Proving stability

For the controlled system

$$\begin{cases} \dot{x} = Ax + Bu \\ u = -Kx \end{cases}$$

to be stable, all real parts of eigenvalues of matrix $[A - BK]$ should be negative. In our case, we have

$$\begin{cases} \dot{x} = \begin{bmatrix} 0 & 1 \\ -15 & -63 \end{bmatrix} x + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u \\ \\ u = -\begin{bmatrix} K_p & K_d \end{bmatrix} e \end{cases}$$

$$A - BK = \begin{bmatrix} 0 & 1 \\ -15 & -63 \end{bmatrix} - \begin{bmatrix} 0 \\ 1 \end{bmatrix} \begin{bmatrix} 2000 & 13 \end{bmatrix}$$

$$A - BK = \begin{bmatrix} 0 & 1 \\ -15 & -63 \end{bmatrix} - \begin{bmatrix} 0 & 0 \\ 2000 & 13 \end{bmatrix}$$

$$A - BK = \begin{bmatrix} 0 & 1 \\ -2015 & -76 \end{bmatrix}$$

Using Matlab's `eig()` function, I obtained the eigenvalues, that are

$$eigenvalues = \begin{bmatrix} -38 + 23.8956i \\ -38 - 23.8956i \end{bmatrix}$$

As it can be seen, they have negative real parts, so the controlled system is stable.

Part D.

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

$$\begin{bmatrix} \dot{x_1} \\ \dot{x_2} \end{bmatrix} = \begin{bmatrix} 10 & 3 \\ 5 & -5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

This system is autonomous MIMO system. First of all, let's check its stability by looking at its eigenvalues, obtained with Python's `numply.linalg.eig()`.

$$eigenvalues = \begin{bmatrix} 10.94 \\ -5.94 \end{bmatrix}$$

As can be seen, it is unstable, because it has eigenvalue with positive real part. Thus, we need to add controller to the system:

$$\dot{x} = Ax + BK_1e_1 + BK_2e_2,$$

where K_1 and K_2 are control matrices for input 1 and 2 respectively, x is state vector, and e_1 and e_2 are two errors from two outputs. Below I assumed that $K_1 = K_2$ and our equation takes the form

$$\dot{x} = Ax + BK(e_1 + e_2),$$

Applying above equation to our case:

$$\begin{bmatrix} \dot{x_1} \\ \dot{x_2} \end{bmatrix} = \begin{bmatrix} 10 & 3 \\ 5 & -5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} K_p & K_d \end{bmatrix} \left(\begin{bmatrix} e_1 \\ \dot{e_1} \end{bmatrix} + \begin{bmatrix} e_2 \\ \dot{e_2} \end{bmatrix} \right)$$

Now let me start implementing it in Python. First, I declared some constants:

```
# Proportional and derivative gain
K_p = 10
K_d = 1
# Final time of integration, and integration step
final_time = 1
step = 0.001
# Our matrices A, B and K
A = np.array([[10,3],[5,-5]])
B = np.array([[1],[1]])
K=np.array([K_p,K_d])
# Desired state x is 0 => stable
x_desired = np.asarray([0, 0])
```

As we have no derivative in state vector, we should somehow calculate it. I propose such formula for the derivative approximation:

$$\dot{x}(t_i) = \frac{x(t_i) - x(t_{i-1})}{t_i - t_{i-1}}$$

As `odeint()` integrates numerically, it can try to compute differential with $\Delta t = t_i - t_{i-1}$ very small (or sometimes 0). That leads to numerical errors, and due to this I changed the formula to

$$\dot{x}(t_i) = \frac{x(t_i) - x(t_{i-1})}{max(t_i - t_{i-1}, 10^{-6})}$$

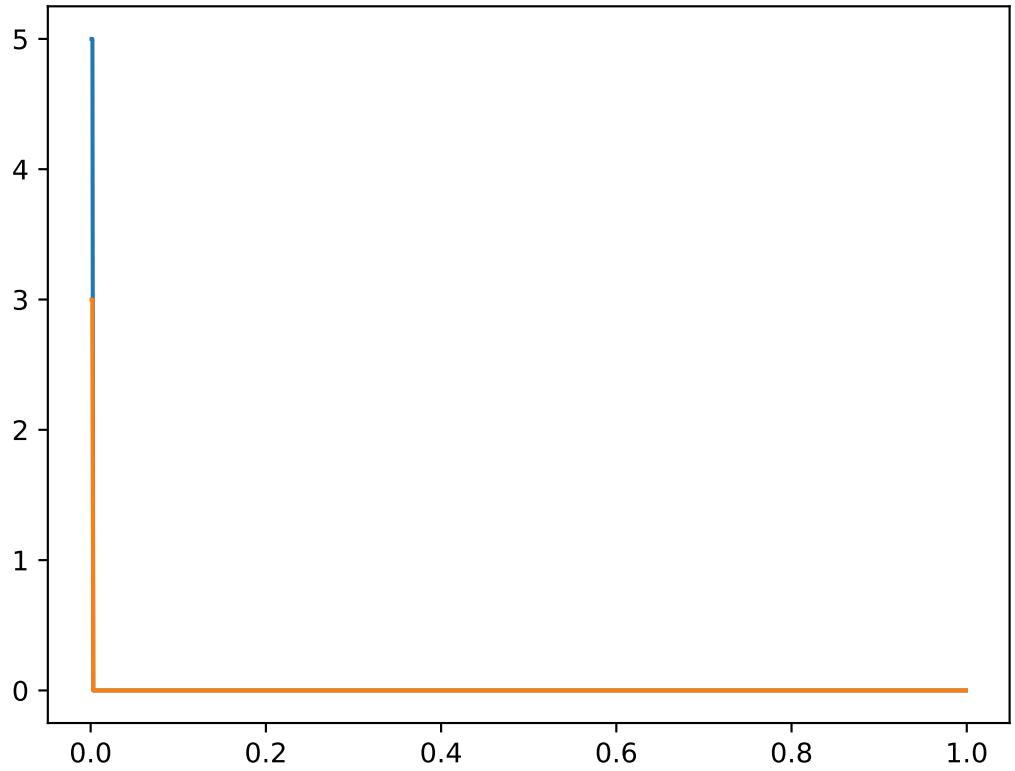
With this assumption, I can now implement differential function:

```
def differential(x,t):
    global last_t, last_error
    error = x_desired - x
    dt = max(t - last_t,1e-6)
    de = error - last_error
    error_dot = de / dt

    error_1  = np.array([error[0],error_dot[0]])
    error_2  = np.array([error[1],error_dot[1]])
    u = K.dot(error_1 + error_2)

    x = x.reshape((2,1))
    x_dot = A.dot(x) + B.dot(u)

    last_t = t
    last_error = error
    x_dot = x_dot.reshape((2,))
    return x_dot
```

Integrating the solution:

```
init = np.array([5,3])
times = np.arange(step, 1, step)
f = plt.figure()
sol = odeint(differential,init,times)
plt.plot(times, sol)
plt.show()
```

The obtained plot of solution is presented below:



As it can be seen, the controller tries to change the system state to desired one in just one time-step. That is not physically possible, and this is always the case, when one uses derivatives of same order as systems themselves, to control the systems.

Part E.

Implement a PI/PID controller for the system: $\ddot{x}+\mu\dot{x}+kx+9.8 = u$ (variants are below) Test your controller on different trajectories, at least two.
For this task, I decided to treat the system as third order one, with $x_1 = \int x dx$, $x_2 = x$, and $x_3 = \dot{x}$. Then, I built a state space model for the system, treating the constant term 9.8 as input.

$$\begin{bmatrix} \dot{x_1} \\ \dot{x_2} \\ \dot{x_3} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & -k & -\mu \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + (-9.8 + u) \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Adding our controller to control the input of the system:

$$\begin{bmatrix} \dot{x_1} \\ \dot{x_2} \\ \dot{x_3} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & -k & -\mu \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} - 9.8 \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} K_i & K_p & K_d \end{bmatrix} \begin{bmatrix} e_{int} \\ e \\ \dot{e} \end{bmatrix}$$

Now let's implement the system on Python. First, I define reference functions:

```
def f_int_des(t):
    return -np.cos(t)

def f_des(t):
    return np.sin(t)

def f_dot_des(t):
    return np.cos(t)
```

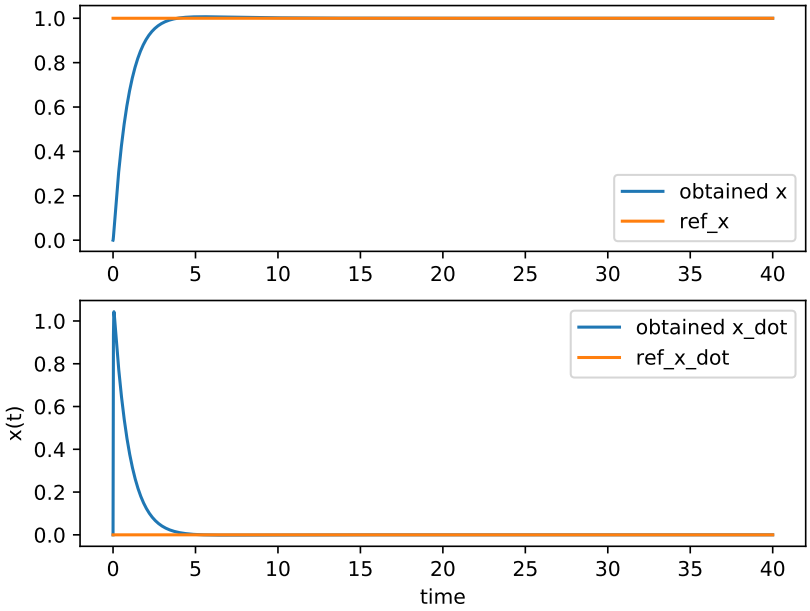
These functions describe our desired trajectory. In our case, $\mu = 63, k = 15$, so I defined them and some other constants:

```
# parameters
mu = 63
k=15
# PID gains
K_p = 100
K_i = 30
K_d = 20
# Final time of integration, and integration step
step = 0.001
final_time = 40
```

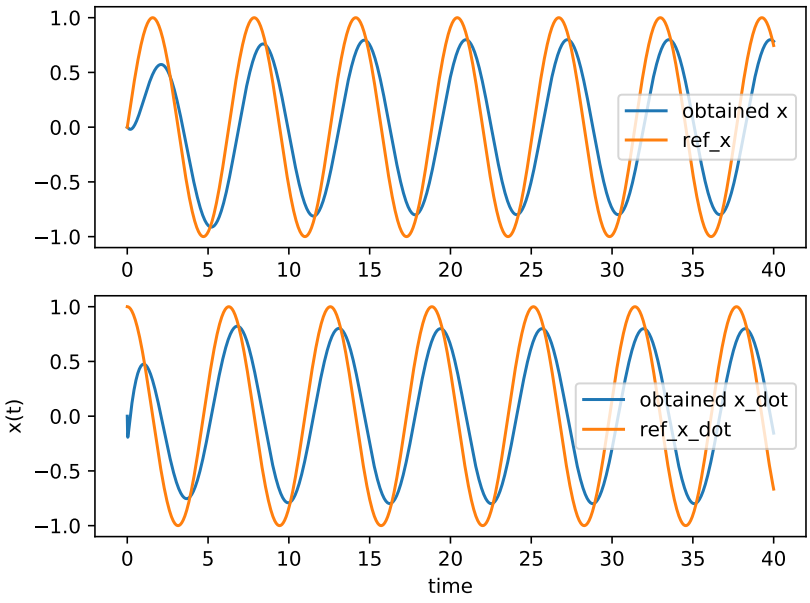
Then I defined differential function:

```
def controlled_system(x,t):
    error_int = f_int_des(t) - x[0]
    error = f_des(t) - x[1]
    error_dot = f_dot_des(t) - x[2]
    u = K_p * error + K_d * error_dot + K_i * error_int
    return np.array([x[1],x[2],u-mu*x[2]-k*x[1]-9.8])
```

As a first trajectory for testing I choose step function $x(t) = 1, \dot{x}(t) = 0$ Here is a plot of its solution with $K_p = 100, K_i = 30, K_d = 20$



Second trajectory is $x(t) = sin(t), \dot{x}(t) = cos(t)$ Here is a plot of its solution with $K_p = 100, K_i = 30, K_d = 20$

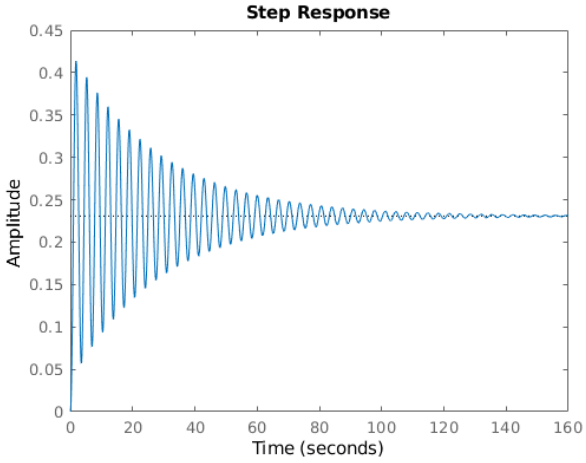


3 Task 3. Design PID controller in Matlab

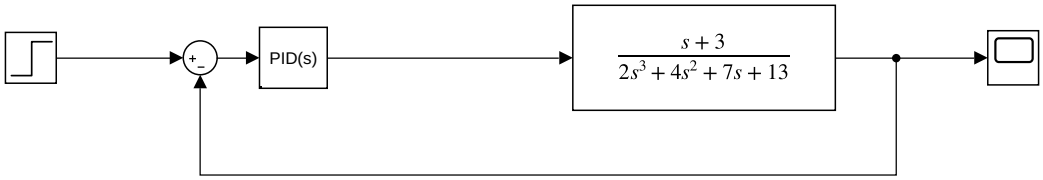
My system:

$$W = \frac{s+3}{2s^3+4s^2+7s+13}$$

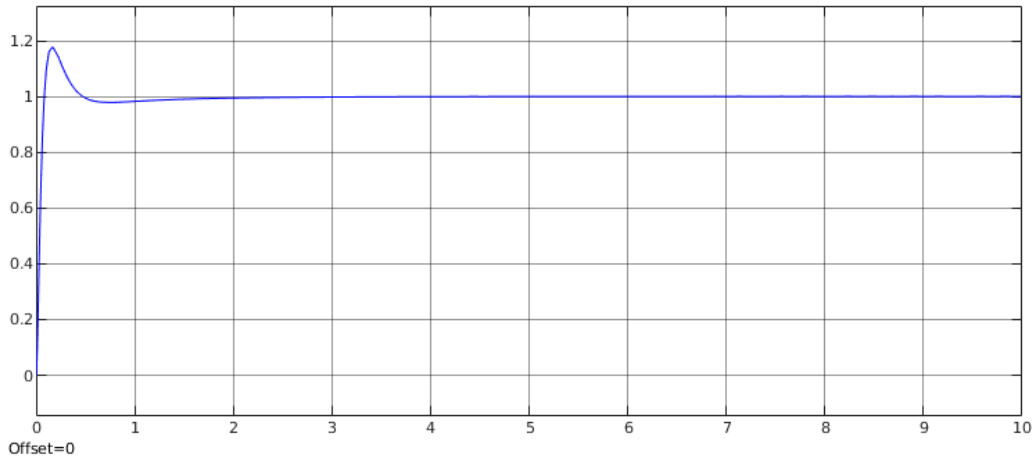
Step response of system without a controller:



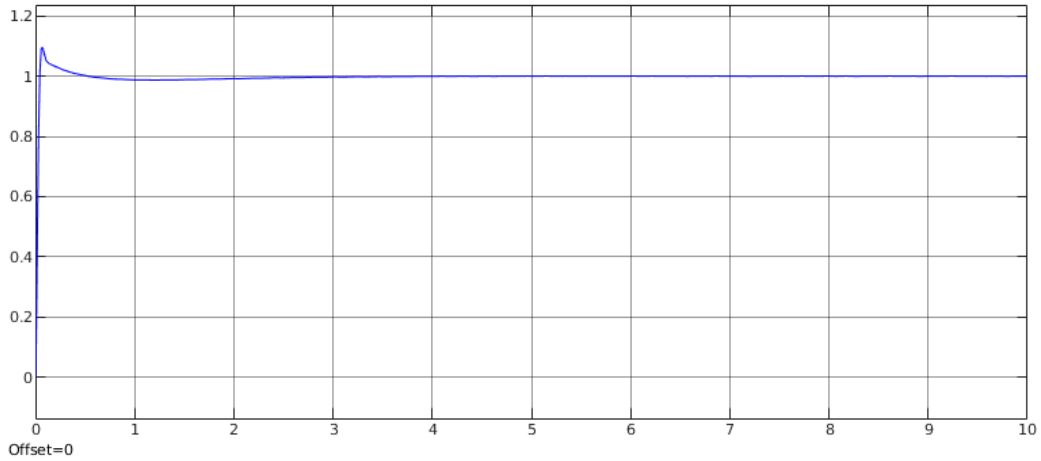
From the plot, it is seen, that system is marginally stable, but has oscillations in the beginning, and there is a steady-state error.
As the first step I have built system in Simulink:



Then, I opened PID controller parameters, and used `PIDtuner` to automatically tune controller gains. The PID tuner have chosen $K_p = 195, K_i = 145, K_d = 20$. (Initial gains were fractional, but I rounded them mathematically).
With these gains, step response start look like this (Note, that time scale has changed!):



After that I started to tune K_d to eliminate overshoot, and obtained minimal overshot of 9.3% with $K_d = 95$.



I decided, that such step response is quite good, and stopped there. The final controller gains are $K_p = 195, K_i = 145, K_d = 20$.

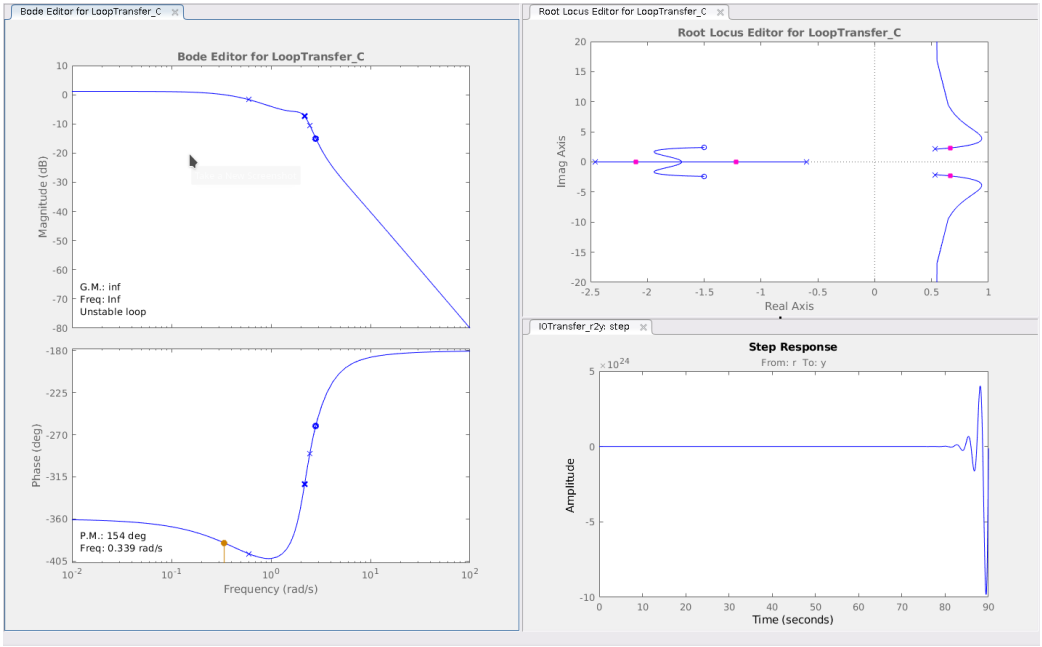
4 Task 4. Design a Lead/Lag compensator in Matlab

My initial system:

$$W(s) = \frac{s^2 + 3s + 8}{s^4 + 2s^3 + 3s^2 + 13s + 7}$$

I entered my plant transfer function to Matlab's console with
`plant = tf([1 3 8],[1 2 3 13 7])`,

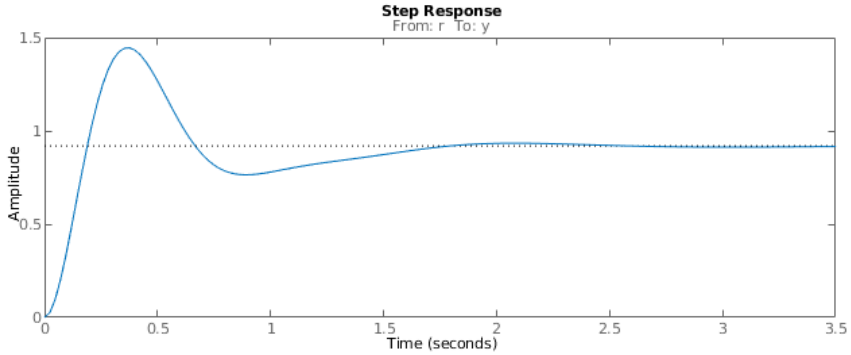
then I opened controlSystemDesigner with
`controlSystemDesigner(plant)`.
From the initial plots it is visible, that my system is not stable.



First of all, we need to improve stability of the system. To do it I will add Lead compensator with pole at -10 and zero at -1 to shift Root-Locus right in the complex plane, and increased compensator gain to 100. My lead compensator formula is now

$$C(s) = 100\frac{s+1}{s+10}$$

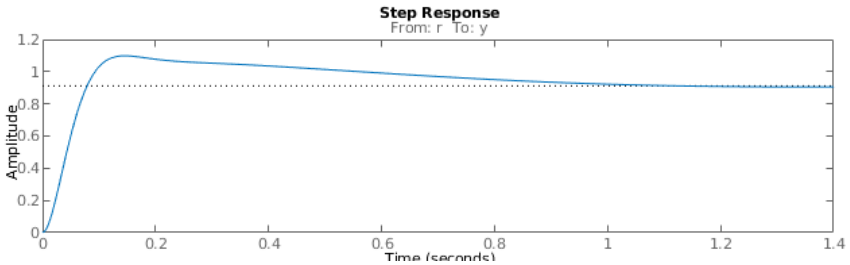
Here is the system step response with untuned Lead Compensator:



Then, I tuned compensator gain, pole and zero and obtained new compensator formula, that has better rise time, and lower overshoot.

$$C(s) = 1125\frac{s + 0.4}{s + 50}$$

The improved step response:



The Lead compensator stabilized the system, give it relatively good rise time, low overshoot, and steady state error is almost not present. Therefore, Lag Compensator for this system is not needed.

5 Used software.

- Python 3.8.1
- Matlab R2018b 9.5.0

All software was run under Manjaro Linux with 5.4.18-rt kernel