CENG483: Behavioural Robotics
Izmir Institute of Technology
Fall 2023
Instructor: Dr. Berat Alper Erol

Homework Set: 06

Ozan Gülbaş

This homework answers the problem set sequentially.

1. *Consider a PUMA 560 6-DOF revolute manipulator*

   (a) *For joint angles:* $\theta_1 = 30$*,* $\theta_2. = 20$*,* $\theta_3 = -35$*,* $\theta_4 = 45$*,* $\theta_5 = 10$*, and* $\theta_6 = 20$ *. Find the final position of the end-effector (x,y,z).*

   First define a instance of a Puma 560 robot using the script:

   ```
   >> mdl_puma560
   ```

   Which creates a 'SerialLink' object, p560, in the workspace.
   We can display the Denavit-Harternberg parameters with the command

   ```
   >> p560

   p560 =

   Puma 560 [Unimation]:: 6 axis, RRRRRR, stdDH, slowRNE
    - viscous friction; params of 8/95;
   +---+-----------+-----------+-----------+-----------+-----------+
   | j |     theta |         d |         a |     alpha |    offset |
   +---+-----------+-----------+-----------+-----------+-----------+
   |  1|         q1|          0|          0|     1.5708|          0|
   |  2|         q2|          0|     0.4318|          0|          0|
   |  3|         q3|    0.15005|     0.0203|    -1.5708|          0|
   |  4|         q4|     0.4318|          0|     1.5708|          0|
   |  5|         q5|          0|          0|    -1.5708|          0|
   |  6|         q6|          0|          0|          0|          0|
   +---+-----------+-----------+-----------+-----------+-----------+
   ```

   The **mdl_puma560** script creates several joint coordinate vectors in the workspace which represent the robot in some canonic configurations:

   - qz (0, 0, 0, 0, 0, 0) ==> zero angle
   - qr (0, pi/2, -pi/2, 0, 0, 0) ==> ready, the arm is straight and vertical
   - qs (0, 0, -pi/2, 0, 0, 0) ==> stretch, the arm is straight and horizontal

- qn (0, pi/4, -pi, 0, pi/4, 0) ==> nominal, the arm is in a dextrous working pose

Given the information above, we need to create a custom joint coordinate vector with the given parameters

```
theta_1 = 30; theta_2 = 20; theta_3 = -35;
theta_4 = 45; theta_5 = 10; theta_6 = 20;

q1 = deg2rad(theta_1); q2 = deg2rad(theta_2);
q3 = deg2rad(theta_3); q4 = deg2rad(theta_4);
q5 = deg2rad(theta_5); q6 = deg2rad(theta_6);

qc = [q1, q2, q3, q4, q5, q6];
```

We can plot a skeleton of the robot with pipes that link coordinate frames as defined by the Denavit-Hartenberg parameters with the command

```
>>  p560.plot(qc)
```

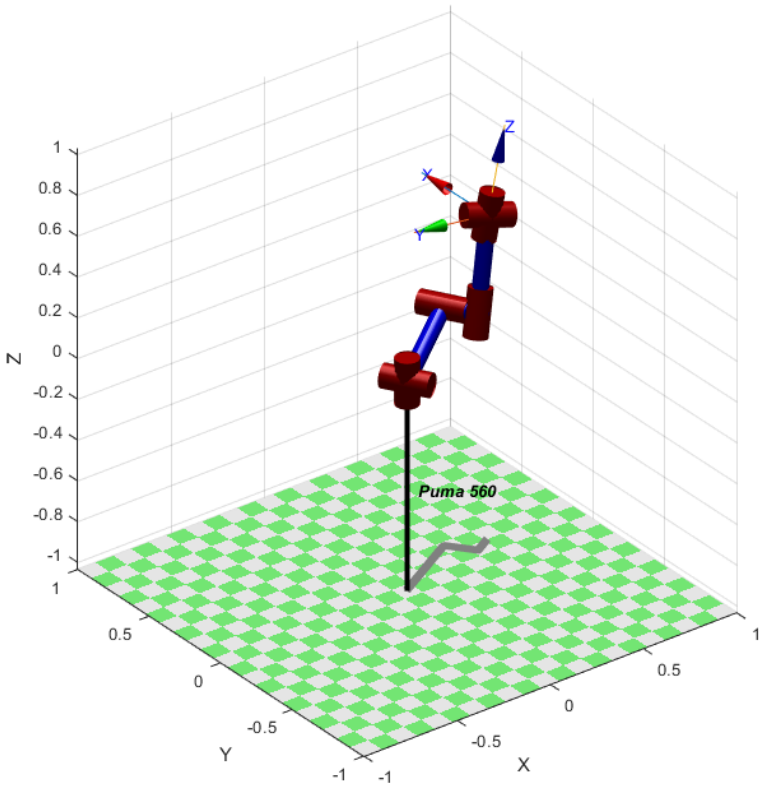

Figure 1: Basic 3D plot of our Puma560 robot with custom joint coordinate vector.

To find the position of the end-effector we can calculate the forward kinematics with **fkine()**.

```
>> TE = p560.fkine(qc)

TE =
   -0.0664   -0.9815    0.1794    0.5402
    0.9965   -0.0744   -0.0382    0.1386
    0.0508    0.1763    0.9830    0.5595
         0         0         0         1
```

(b) *For a position (x,y,z) = (0.1, 0.5, -0.25) in task space, find all joint angles that can achieve such an end point in joint space of the Puma.*

Since the Puma 560 is a 6-axis robot arm with a spherical wrist we use the method **ikine6s()** to compute the inverse kinematics using a closed-form solution. First, we need to create a 4x4 homogenous transform matrix with **SE3()** function.

```
T = SE3([0.1, 0.5, -0.25]);
```

Then, find the joint angles for `T`:

```
>> Q = p560.ikine6s(T)

Q =
    4.2163   -3.5525    0.2447    3.1416    2.9754   -1.0747
```

(c) *Use the initial homogeneous transformation matrix of part (a) as initial position of the Puma and the final homogeneous matrix of part (b) as final position. Use the toolbox to create a joint trajectory using any of the trajectory generation commands in the toolbox.*

Initial position homogenous transformation matrix TI:

```
TI = p560.fkine(qc);
```

Final postion homogenous transformation matrix TF:

```
TF = T;
```

The associated joint coordinate vectors:

```
qi = p560.ikine6s(TI);
qf = p560.ikine6s(TF);
```

And we require the motion to occur over a time period of 2 seconds in 50ms time steps

```
t = [0:0.05:2]';
```

A joint-space trajectory is formed smoothly interpolating between the joint configurations `qi` and `qf`. I used the **tploy()** scalar interpolation function with the multi-axis driver function **mtraj()**.

```
>> trajectory = mtraj(@tpoly, qi, qf, t)

trajectory =

    3.1204    1.8790   -0.6109   -3.1028    1.0872    1.6482
    3.1205    1.8782   -0.6107   -3.1018    1.0875    1.6478
    3.1216    1.8727   -0.6099   -3.0955    1.0894    1.6450
    3.1245    1.8586   -0.6076   -3.0793    1.0943    1.6380
    3.1297    1.8325   -0.6035   -3.0493    1.1034    1.6249
    3.1380    1.7918   -0.5971   -3.0025    1.1175    1.6045
    3.1495    1.7345   -0.5881   -2.9366    1.1374    1.5757
    3.1648    1.6590   -0.5762   -2.8498    1.1637    1.5379
    3.1838    1.5644   -0.5613   -2.7411    1.1966    1.4905
    3.2069    1.4504   -0.5433   -2.6099    1.2362    1.4333
    3.2338    1.3168   -0.5223   -2.4564    1.2826    1.3663
    3.2646    1.1641   -0.4983   -2.2809    1.3357    1.2898
    3.2991    0.9933   -0.4713   -2.0844    1.3951    1.2041
    3.3370    0.8053   -0.4417   -1.8683    1.4605    1.1099
    3.3781    0.6017   -0.4097   -1.6343    1.5312    1.0078
    3.4220    0.3842   -0.3754   -1.3843    1.6068    0.8988
    3.4683    0.1548   -0.3393   -1.1205    1.6866    0.7838
    3.5165   -0.0843   -0.3016   -0.8456    1.7697    0.6639
    3.5663   -0.3309   -0.2628   -0.5621    1.8554    0.5403
    3.6170   -0.5826   -0.2231   -0.2728    1.9429    0.4142
    3.6683   -0.8367   -0.1831    0.0194    2.0313    0.2867
    3.7196   -1.0909   -0.1430    0.3116    2.1196    0.1593
    3.7704   -1.3426   -0.1034    0.6009    2.2071    0.0332
    3.8202   -1.5892   -0.0646    0.8844    2.2928   -0.0905
    3.8684   -1.8283   -0.0269    1.1594    2.3760   -0.2104
    3.9147   -2.0577    0.0092    1.4231    2.4557   -0.3253
    3.9586   -2.2752    0.0435    1.6731    2.5313   -0.4344
    3.9996   -2.4788    0.0756    1.9072    2.6021   -0.5364
    4.0376   -2.6667    0.1052    2.1233    2.6674   -0.6307
    4.0721   -2.8376    0.1321    2.3197    2.7268   -0.7163
    4.1029   -2.9903    0.1561    2.4952    2.7799   -0.7928
    4.1298   -3.1239    0.1772    2.6488    2.8263   -0.8598
    4.1528   -3.2379    0.1951    2.7799    2.8660   -0.9170
    4.1719   -3.3325    0.2100    2.8886    2.8989   -0.9644
    4.1871   -3.4080    0.2219    2.9754    2.9251   -1.0022
    4.1987   -3.4653    0.2310    3.0414    2.9450   -1.0310
    4.2069   -3.5060    0.2374    3.0881    2.9592   -1.0514
    4.2122   -3.5321    0.2415    3.1181    2.9683   -1.0645
    4.2150   -3.5462    0.2437    3.1344    2.9732   -1.0716
    4.2161   -3.5517    0.2446    3.1407    2.9751   -1.0743
    4.2163   -3.5525    0.2447    3.1416    2.9754   -1.0747
```

(d) *Using the initial joint angles of* $\theta = [0, 90, 25, 0, 30, 20]$*, T and a torque vector of* $\tau = [1, 0, 0, 0, 0, 0.5]$ *simulate the Puma 560 on the toolbox and plot the 3-D trajectory of the robot for 5 different instances of time in the range of* $0 \leq t \leq 10$.

For the Puma 560 robot in its initial pose with the joint angles of $\theta = [0, 90, 25, 0, 30, 20]$, and a torque vector of $\tau = [1, 0, 0, 0, 0, 0.5]$.

```
qt = [
    deg2rad(0),
    deg2rad(90),
    deg2rad(25),
    deg2rad(0),
    deg2rad(30),
    deg2rad(20)
];
tau = p560.jacob0(qt)' * [1, 0, 0, 0, 0, 0.5]';
```

Create a time period of 8 seconds in 2s time steps because we need 5 instances in time.

```
t = [0:2:8]';
```

Calculate the joint-space trajectory of the robot.

```
trajectory = mtraj(@tpoly, qt, tau', t);
```

Simulate the Puma 560

```
p560.plot(trajectory);
```

To plot the trajectory of the Puma 560 robot in Cartesian space we need to apply forward kinematics first and take the translational part of this trajectory because the resulting matrix of **fkine()** is an array of **SE3** objects.

```
TT = p560.fkine(trajectory);
p = TT.transl;
plot(p);
xlabel("Time (s)"); ylabel("Position (m)"); legend("x", "y", "z");
```

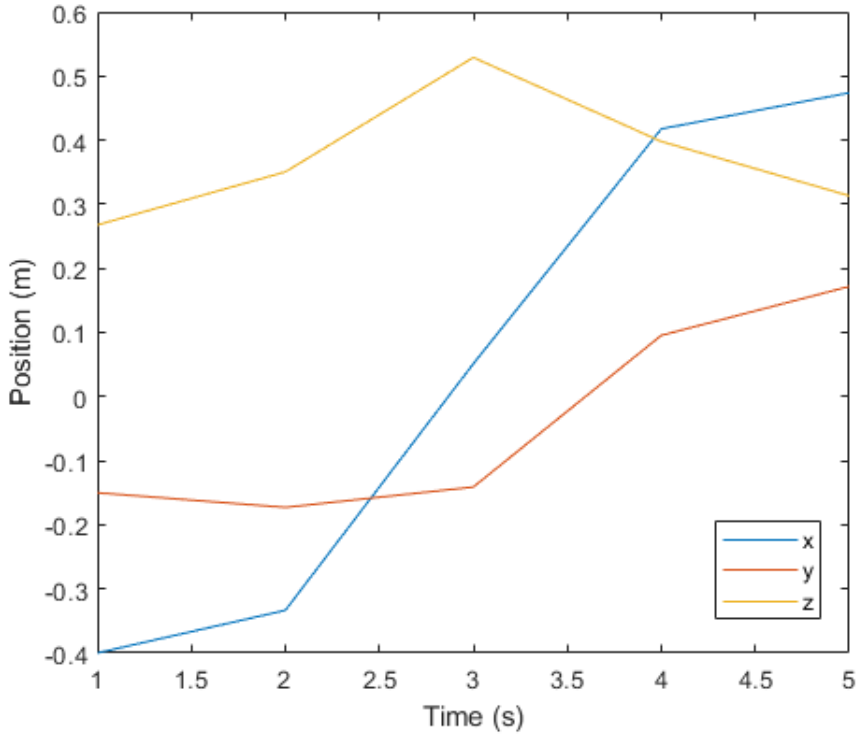

Figure 2: Plot of the trajectory in Cartesian space in 5 different instances of time.

2. *Consider the* ***sl_ driveline*** *command in section 4.2.2. We wish to create a map of* 10×10 *like Figure 4.9 and draw 2 lines as* $y = -2x + 10$ *and* $y = 2x - 10$ *on this map. These lines will divide the* (x, y) *plane into 3 segments. Use Simulink and bicycle command and at least one pose in each of the 3 regions. Discuss the results and justify your answers.*

   Firstly, we need to start the Simulink model

   ```
   >> sl_driveline
   ```

   After we started the Simulink model, we specify 2 target lines $y_1 = -2x_1 + 10$ and $y_2 = 2x_2 - 10$. We need to specify L considering $L = [A, B, C]$ where $Ax + By + C = 0$.

   ```
   % Run it with the L = [2, 1 -10] also
   L = [-2, 1, 10];
   x = [0:10];
   y1 = -2*x + 10;
   y2 = 2*x - 10;
   ```

   Then we need to define 3 poses for each 3 of the regions and plot the results.

   ```
   % pose : [x, y, angle]
   poses = [9, 3, pi/2];
   poses(:, :, 2) = [2, 2, 0];
   poses(:, :, 3) = [7, 5, -pi/2];
   ```

```
for i = 1 : 3
    x0 = poses(:, :, i);

    % Finally simulate the motion
    r = sim('sl_driveline');

    % Extract the y as a function of time
    q = r.find('y');

    %Plot and hold the trajectories
    plot(q(:,1),q(:,2), "-^", "MarkerIndices", [1], "MarkerSize",15);
    grid on; xlim([0,10]); ylim([0,10]);
    hold on; plot(x, y1, "--"); hold on; plot(x, y2, "--");
    xlabel("x"); ylabel("y");

end
```

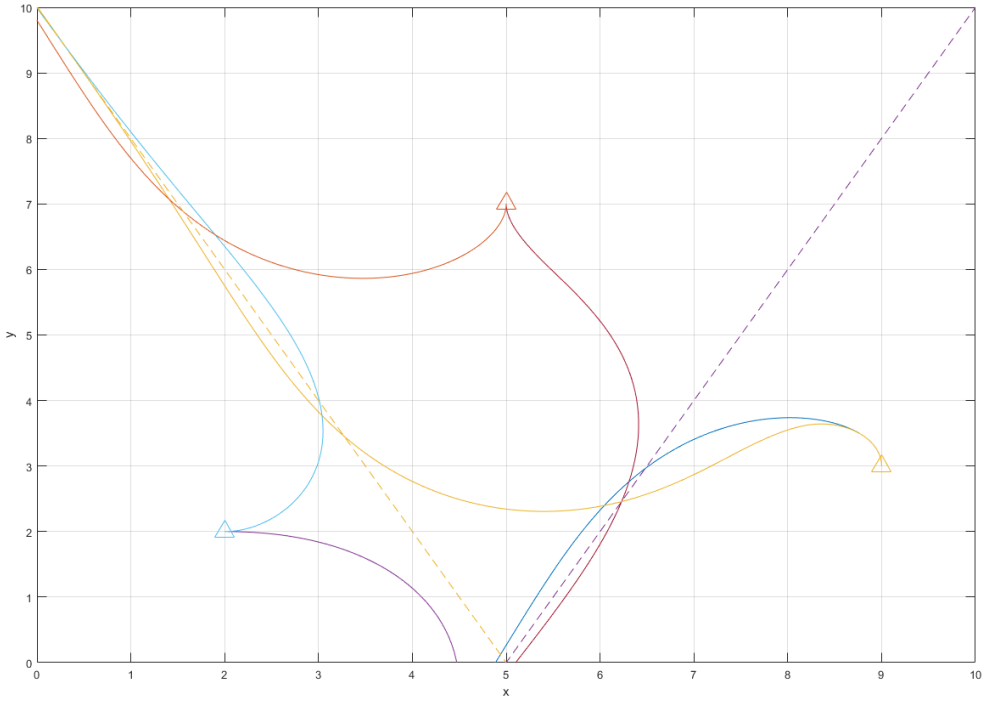

Figure 3: Plot of the simulation results with **hold on**.

As we can see from the Figure 3, each point with a position converges to the selected line which is the expected outcome of **sl_driveline** method.

3. *Consider the **sl_pursuit** command in section 4.2.3. We wish to create a map of 10 x 10 like Figure 4.11a and use 4 initial poses from the 4 corners of the map and one pose inside the circular path. Plot all x-y and speed vs time trajectories. Discuss the results and justify your answers.*

   I could not have the time to work on this question unfortunately.

4. *Consider the flying robots section 4.3. Repeat exercise starting from page 83. Repeat the exercise and use different set of initial conditions and scenarios in* ***sl_ quadrotor*** *command. Discuss the results and justify your answers.*

   First load the complete control system for a quadrator with

   ```
   >> sl_quadrotor
   ```

   Then we can load the parameters of a specific quadrator with

   ```
   >> mdl_quadrotor
   ```

   Which creates a structure called quadrotor in the workspace, and its elements are the various dynamic properties of the quadrotor.

   We can run the simulation using

   ```
   >> sim('sl_quadrotor')
   ```

   To plot x and y versus time

   ```
   >> plot(result(:,1), result(:,2:3))
   ```

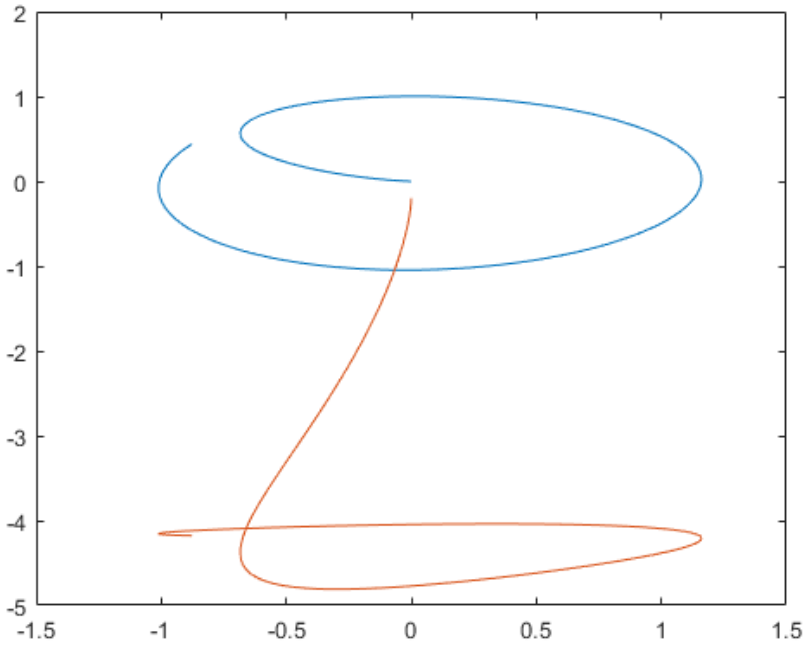

Figure 4: Plot of the quadrotor x and y versus time with default values.

As we can see in Figure 5 changing z* from -4 to -2 which is the input of the height control system, shrinks y vs time by half, causing the change of limits from [-5, 0] to [-2.5, 0]. Changing the frequency of the x* and y* input signals, causes absurd changes in the graph that I cannot comment properly.

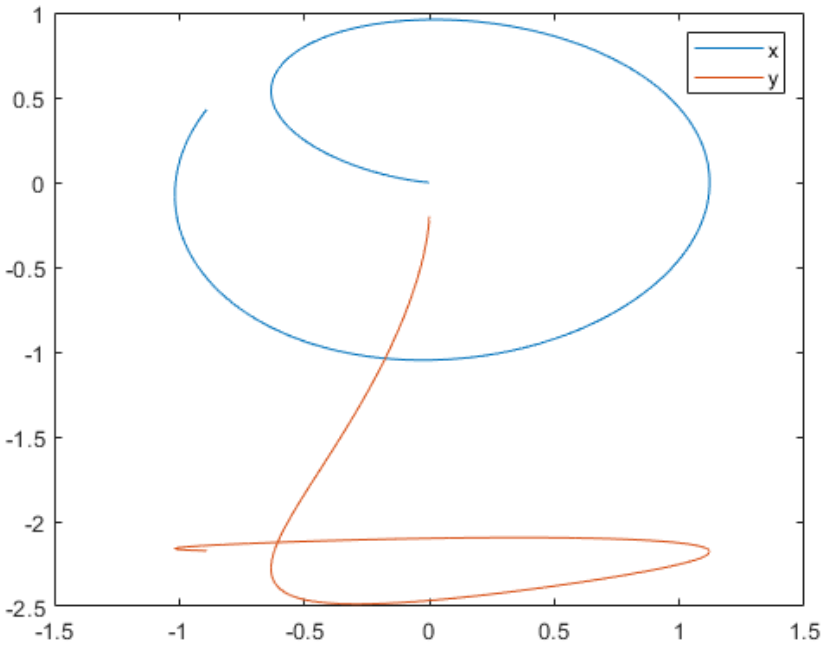

Figure 5: Plot of the quadrotor x and y versus time with z* = -2 instead of -4.

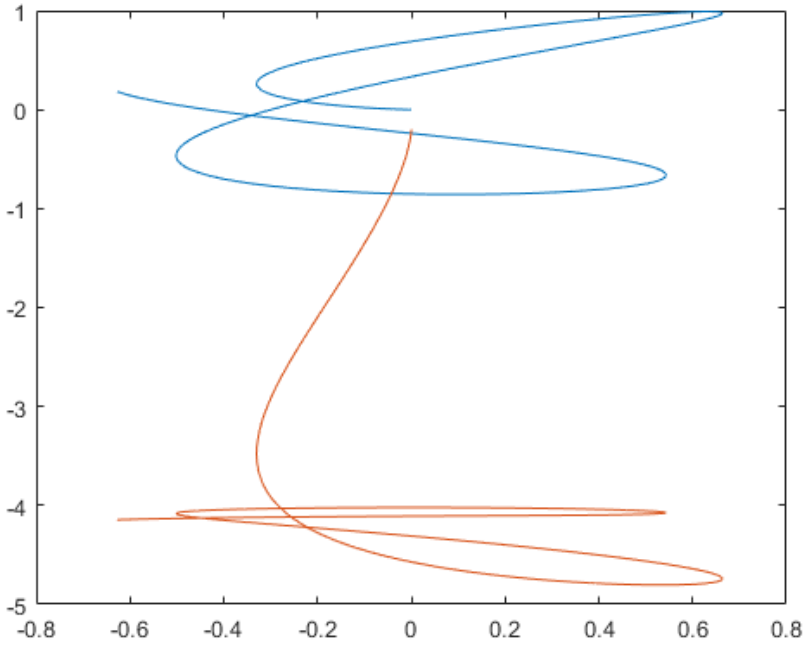

Figure 6: Plot of the quadrotor x and y versus time with input x* sinus function frequency component doubled.

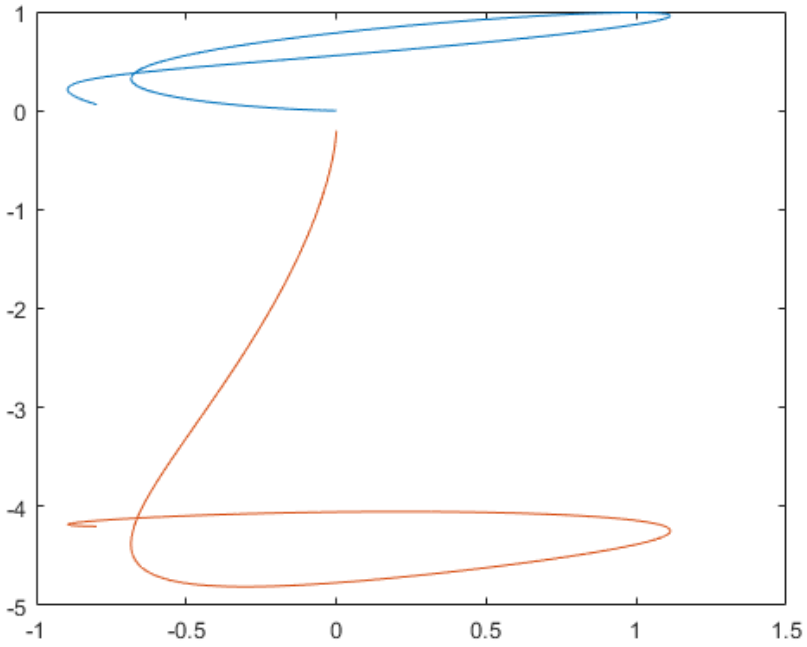

Figure 7: Plot of the quadrotor x and y versus time with input y* sinus function frequency component cut by half.

1 References

Corke, P. (2017). Robotics, vision and control fundamental algorithms in MATLAB. Springer International Publishing.

Cem Tolga Münyas, Boğaçhan Ali Düşgül