UFME7R-15-M Robot Learning and Teleoperation Coursework

Report Template: Develop a Method for Single-Arm Robot Manipulation

```
clearvars
```

How do I use this file?

This MLX file should be how you put together your main submission for this coursework. Please read the coursework brief on Blackboard and complete this MLX along with the class files, `SkillGeneralisation.m` and `MixtureGaussian.m` to have a full coursework submission.

Your final submission should consist of this MLX and an exported PDF of this MLX, and your `SkillGeneralisation.m` and `MixtureGaussian.m`. If you are unsure of how to use an MLX, or how to export for submission, please see the guide on Blackboard: https://blackboard.uwe.ac.uk/bbcswebdav/pid-11024632-dt-content-rid-53216638_2/xid-53216638_2

You should keep all your experimental results and report document within this MLX. However, to make the code work in this MLX, you need to complete some functions in the `SkillGeneralisation.m` and `MixtureGaussian.m` as well. Please read the read the instructions carefully before start your work. If you are proficient with writing MATLAB functions and wish to use them to organise your code, please define them as static functions/methods in `RobotLearning.m` rather than declaring them as separate files. If for some reason this isn't possible, please submit any additional files alongside.

Please also note the following:

- There are some outputs in the template, which are used as reference. Please delete them before you start to add your content.
- Your PDF exported from this MLX will be marked and checked with your completed class files.
- Very wide images do not render well onto the PDF. Please check to make sure images you insert are captured appropriately within the PDF.

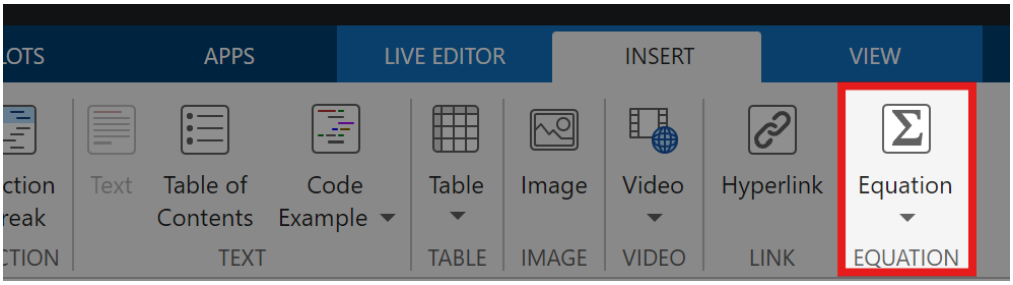
Task 1 (10 Marks)

- Utilise Dynamic Movement Primitives (DMPs) alongside Gaussian Mixture Models (GMM) and Gaussian Mixture Regression (GMR) to devise an enhanced Dynamic Movement Primitives for single-arm robot manipulation.
- During the development of the new algorithm, you can refer to the paper "DMP and GMR based Teaching by Demonstration for a KUKA LBR Robot" by Hewitt et al., introduced in the reading lecture.
- This paper is included on Blackboard: https://blackboard.uwe.ac.uk/bbcswebdav/pid-11172103-dt-content-rid-53151419_2/xid-53151419_2
- Write an introduction about the calculation steps based on your understanding of this paper, into your own report.

Useful materials for this Part include:

- Lecture notes on DMPs: https://blackboard.uwe.ac.uk/bbcswebdav/pid-11169718-dt-content-rid-53121141_2/xid-53121141_2
- Lecture notes on Linear models: https://blackboard.uwe.ac.uk/bbcswebdav/pid-11024325-dt-content-rid-52100419_2/xid-52100419_2
- Lecture notes on GMM: https://blackboard.uwe.ac.uk/bbcswebdav/pid-11065266-dt-content-rid-52179295_2/xid-52179295_2
- Lecture notes on GMR: https://blackboard.uwe.ac.uk/bbcswebdav/pid-11151991-dt-content-rid-52934847_2/xid-52934847_2

Describe your calculations below (max 500 words):



This section will explain how to use human demonstrations to teach a single-arm robot new motion tasks. We model these motions as DMPs and enhance the system using GMM-GMR mentioned in Hewitt et al., 2017 to learn trajectories from Ndemonstrations of end-effector trajectories $(x_t, \dot{x}_t, \ddot{x}_t)$ over time.

Let's start with one of the component of DMP, which is a canonical dynamical system given by.

$$\tau \dot{s} = -\alpha_s s$$

where $s(t) \in [1, 0]$ denotes the states of the canonical system

Another components of DMP is transformed system r. It consists of two parts: a spring-damper system and a nonlinear forcing term, $f(s)$, in Cartesian Space.

$$\tau \dot{v} = \alpha(\beta(g - x) - v) + (g - x_0) f(s)$$

$$\tau \dot{x} = v$$

where τ denotes the temporal scaling factor, α, β denote that gain terms for damping and spring coefficient respectively, $x \in R$ is the position of the end-effector, $v \in R$ is the velocity of the end-effector, g is the goal, and x_0 denotes the initial position.

Instead of defining the forcing term $f(s)$ from the demonstrations using normalized radial basis functions (RBF), we model the nonlinear transformation function using GMM.

For a set of N demonstration trajectories $\left\{x_t^n, \dot{x}_t^n, \ddot{x}_t^n\right\}_{t=0,n=1}^{T^n, N}$, compute the forcing term f_t at each time step t.

$$f_t = \frac{\tau \dot{v}_t - \alpha(\beta(g - x_t) - v_t)}{(g - x_0)};$$

Given the joint probability distribution:

$$p(s, f) = \sum_{k=1}^{K} \pi_k \cdot \mathrm{N}\left(\begin{bmatrix} s \\ f \end{bmatrix}; \mu_k, \Sigma_k\right)$$

$$\mu_k = \begin{bmatrix} \mu_{s,k} \\ \mu_{f,k} \end{bmatrix},\ \Sigma_k = \begin{bmatrix} \Sigma_{s,k} & \Sigma_{\mathrm{sf},k} \\ \Sigma_{\mathrm{fs},k} & \Sigma_{f,k} \end{bmatrix}$$

$$\sum_{k=1}^{K} \pi_k = 1$$

where Kdenotes the number of components, π_k denotes mixture weight (priors) of the k-th component, μ_k is mean vectors of the k-th component, Σ_k is covariance matrices of the k-th component, and $\mathrm{N}\left(\begin{bmatrix} s \\ f \end{bmatrix}; \mu_k, \Sigma_k\right)$ is Gaussian probabilty distribution N is

$$\mathrm{N}\left(\begin{bmatrix} s \\ f \end{bmatrix}; \mu_k, \Sigma_k\right) = \frac{e^{-0.5\left(\begin{bmatrix} s \\ f \end{bmatrix} - \mu_k\right)^T \Sigma_k^{-1} \left(\begin{bmatrix} s \\ f \end{bmatrix} - \mu_k\right)}}{2\pi \sqrt{|\Sigma_k|}}\mathrm{t}$$

We start by initialising μ_k using K-means clustering, setting $\pi_k = \frac{1}{K}$, and setting each Σ_k identity matrices. The parameters in GMM model can be estimated using Expectation-Maximization (EM) algorithm. For each iteration, we compute **E-step** (Expectation) to update responsibilities γ_{nk}, and update the parameters using **M-Step** (Maximisation) as follows.

E-step:

$$\gamma_{\mathrm{nk}} = \frac{\pi_k \cdot \mathrm{N}(x_t; \mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \cdot \mathrm{N}(x_t; \mu_j, \Sigma_j)}$$

M-step:

$$\text{update prior} : \pi_k = \frac{\sum_{t=1}^{T} \gamma_{\mathrm{tk}}}{T}$$

$$\text{update mean} : \mu_k = \frac{1}{\sum_{t=1}^{T} \gamma_{\mathrm{tk}}} \left(\sum_{t=1}^{T} \gamma_{\mathrm{tk}} \cdot x_t\right)$$

$$\text{update covariances}: \Sigma_k = \frac{1}{\sum_{t=1}^{T} \gamma_{\text{tk}}} \left(\sum_{t=1}^{T} \gamma_{\text{tk}} \cdot (x_t - \mu_k) \cdot (x_t - \mu_k)^T \right)$$

Then, repeat **E-step** and **M-step** until log-likelihood converges:

$$L = \sum_{t=1}^{T} \log \left(\sum_{k=1}^{K} \pi_k \cdot \mathrm{N}(x_t; \mu_k, \Sigma_k) \right)$$

The iteration stops when the change in log-likelihood is below a threshold or a maximum number of iterations is reached.

After the parameters are obtained, we predict $\widehat{f}(s)$ using Gaussian Mixture Regression (GMR). Now, to use the GMM for DMP execution, apply GMR to get the expected value of f conditioned on s: GMR Conditional Mean is given by.

$$\widehat{f}_s = \sum_{k=1}^{K} h_k(s) \left[\mu_{f,k} + \Sigma_{\text{fs},k} \, (\Sigma_{s,k})^{-1} (s - \mu_{s,k}) \right]$$

where $h_k(s)$is responsibility of component k for input s.

$$h_k(s) = \frac{\pi_k \cdot \mathrm{N}(s; \mu_{s,k}, \Sigma_{s,k})}{\sum_{j=1}^{K} \pi_j \cdot \mathrm{N}(s; \mu_{s,j}, \Sigma_{s,j})}$$

where $\mu_{f,k}, \mu_{s,k}$ is means of s and $f$in component k, and $\Sigma_{\text{fs},k}, \Sigma_{s,k}$ is sub-blocks of the covariance matrix.

By using GMR method, we can define the forcing term for learning by employing DMP.

References

[1] Hewitt, A., Yang, C., Li, Y. and Cui, R., 2017, September. DMP and GMR based teaching by demonstration for a KUKA LBR robot. In *2017 23rd International Conference on Automation and Computing (ICAC)* (pp. 1-6). IEEE.

Task 2 (50 marks)

Implement your improved DMP method using the following steps:

Part 1 (15 marks):

- Employ the original DMP to compute the values of nonlinear term (the forcing term) of the inputs.

Useful materials for this Part include:

- Lecture notes on DMPs: https://blackboard.uwe.ac.uk/bbcswebdav/pid-11169718-dt-content-rid-53121141_2/xid-53121141_2
- Code examples for DMPs: https://blackboard.uwe.ac.uk/webapps/blackboard/content/listContentEditable.jsp?content_id=_11169981_1&course_id=_365949_1
- Documentation for loading data into MATLAB: https://uk.mathworks.com/help/matlab/ref/load.html

The below code provides a starting point for Part 1. You need to complete the functions in the `SkillGeneralisation.m` to make the code work.

Dataset parameters

```
nbData = 200;                % Length of each demo trajectory
nbDemos = 5;                 % Number of demonstrations for the DMP training
```

Define DMP parameters

```
nStates = 9;                 % Number of activation functions (i.e., number
of states in the forcing term)
nVar = 3;                    % Number of variables [s,f1,f2] (decay term and
perturbing force)
beta = 50;                   % Stiffness gain (β)
alpha = (2*beta)^.3;         % Damping gain (with ideal underdamped damping
ratio) (α)
decayFactor = 1.1;           % Decay factor (a in τx'=-ax)
dt = 0.01;                   % Duration of time step (τ)
```

Load handwriting trajectories

```
demos = [];
load('G.mat');
demos_in = {demos{1:nbDemos}};
```

Initialise the DMP instance

```
dmp = SkillGeneralisation(nVar, nStates, beta, alpha, dt);   % Class
constructor
dmp = dmp.canonicalSystemInitialisation(decayFactor, nbData);
dmp = dmp.trajectort2Forcing(demos_in);
```

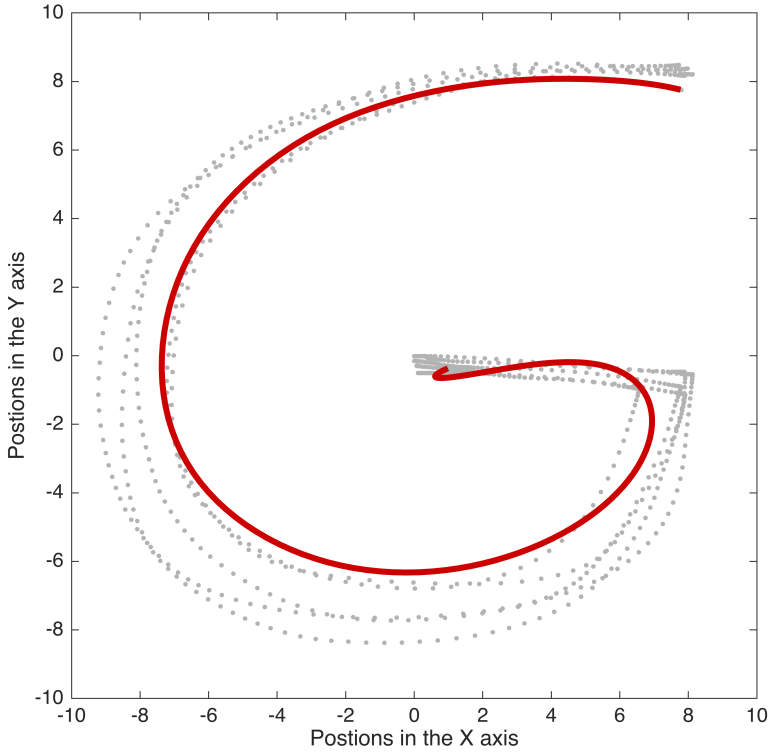
Regression with Locally Weighted Linear Regression (WLR)

```
fWLR = dmp.fittingWithLocallyWLS();
trajWLR = dmp.forcing2Trajectory(fWLR);
```

Spatial plot

```
figure,
for n = 1:nbDemos
```

```
    plot(demos_in{n}.pos(1,:),
demos_in{n}.pos(2,:),'.','markersize',8,'color',[.7 .7 .7]); hold on
end
plot(trajWLR(1,:),trajWLR(2,:),'-','linewidth',3,'color',[.8 0 0]);
xlabel('Postions in the X axis')
ylabel('Postions in the Y axis')
axis equal; axis square;
```



Part 2 (15 marks):

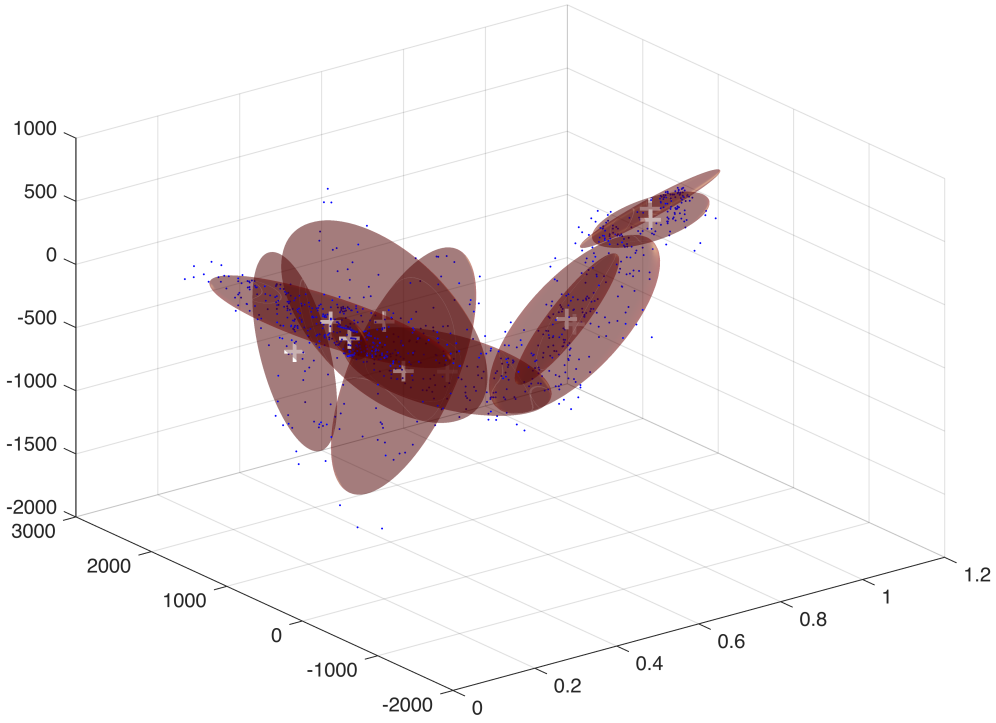
- Utilize Gaussian Mixture Models (GMM) and Gaussian Mixture Regression (GMR) instead of conventional Locally Weighted Linear Regression (WLR) to calculate the trajectory.

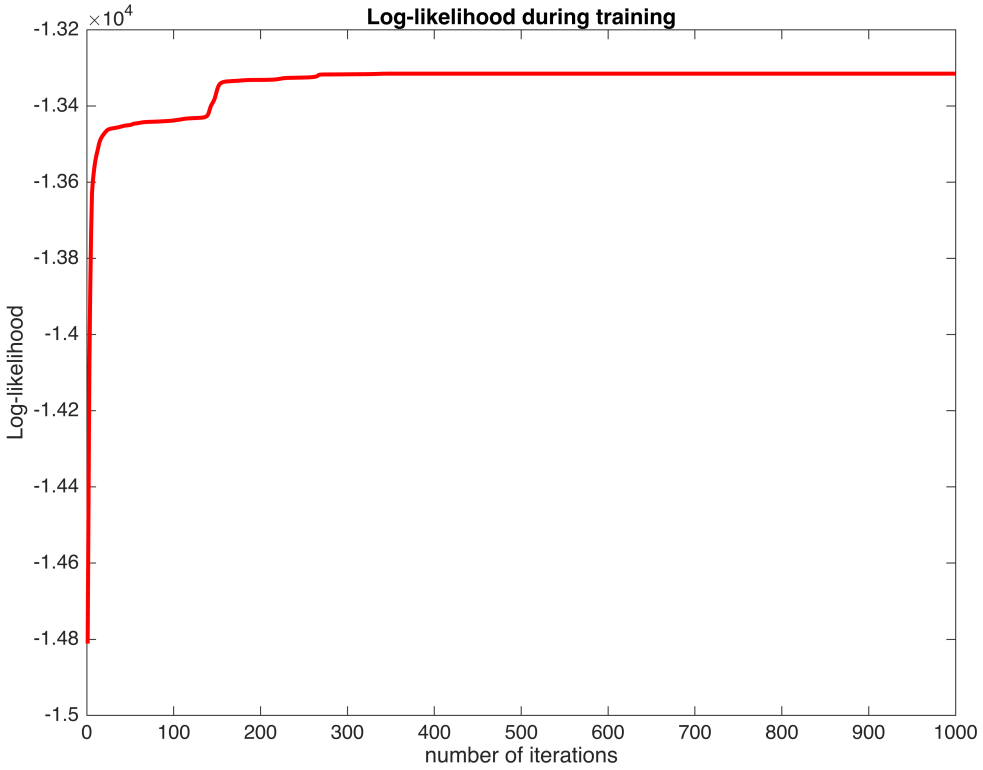
Useful materials for this Part include:

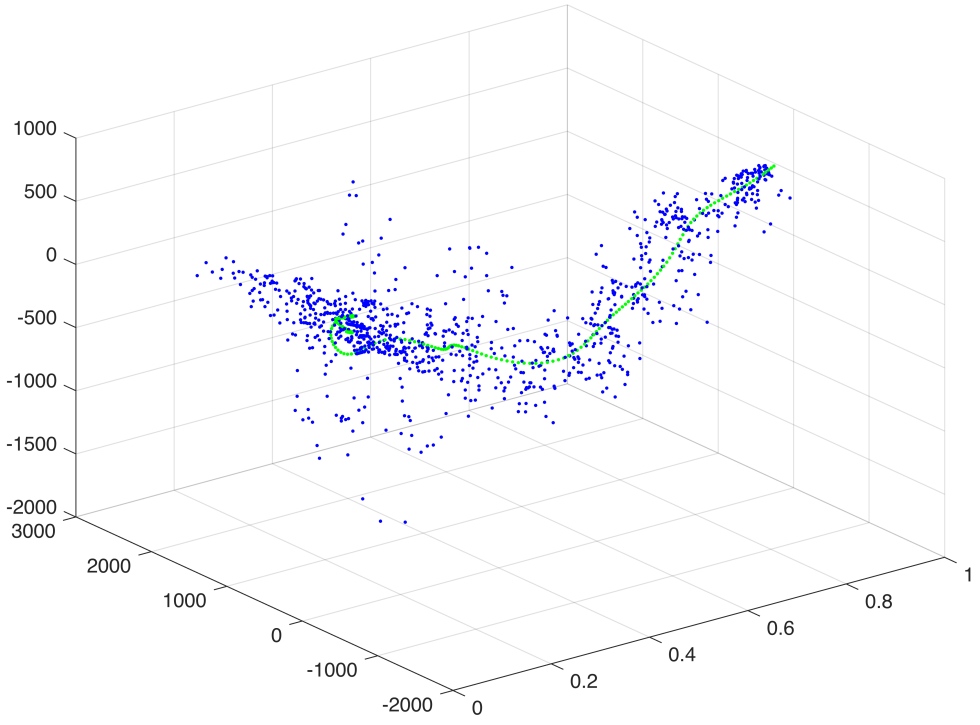
- Lecture notes on GMM: https://blackboard.uwe.ac.uk/bbcswebdav/pid-11065266-dt-content-rid-52179295_2/xid-52179295_2
- Lecture notes on GMR: https://blackboard.uwe.ac.uk/bbcswebdav/pid-11151991-dt-content-rid-52934847_2/xid-52934847_2
- Lab tutorial notes on GMM: https://blackboard.uwe.ac.uk/webapps/blackboard/content/listContentEditable.jsp?content_id=_11071979_1&course_id=_365949_1
- Lab tutorial notes on GMR: https://blackboard.uwe.ac.uk/webapps/blackboard/content/listContentEditable.jsp?content_id=_11151988_1&course_id=_365949_1

The below code provides a starting point for Part 2. You need to complete the functions in the `MixtureGaussian.m` to make the code work.

```
fGMR = dmp.fittingWithGMR();
```



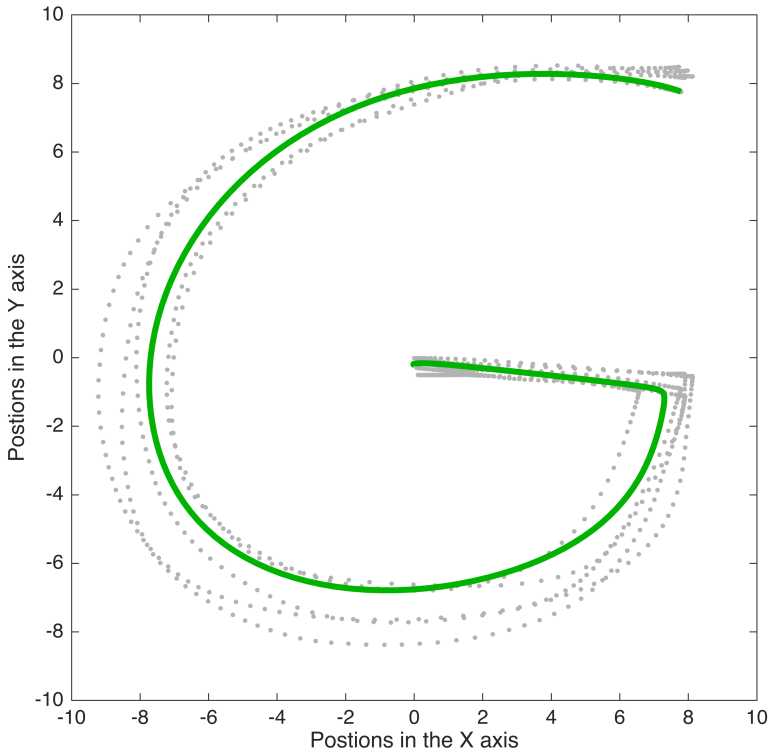





```
trajGMR = dmp.forcing2Trajectory(fGMR);
```

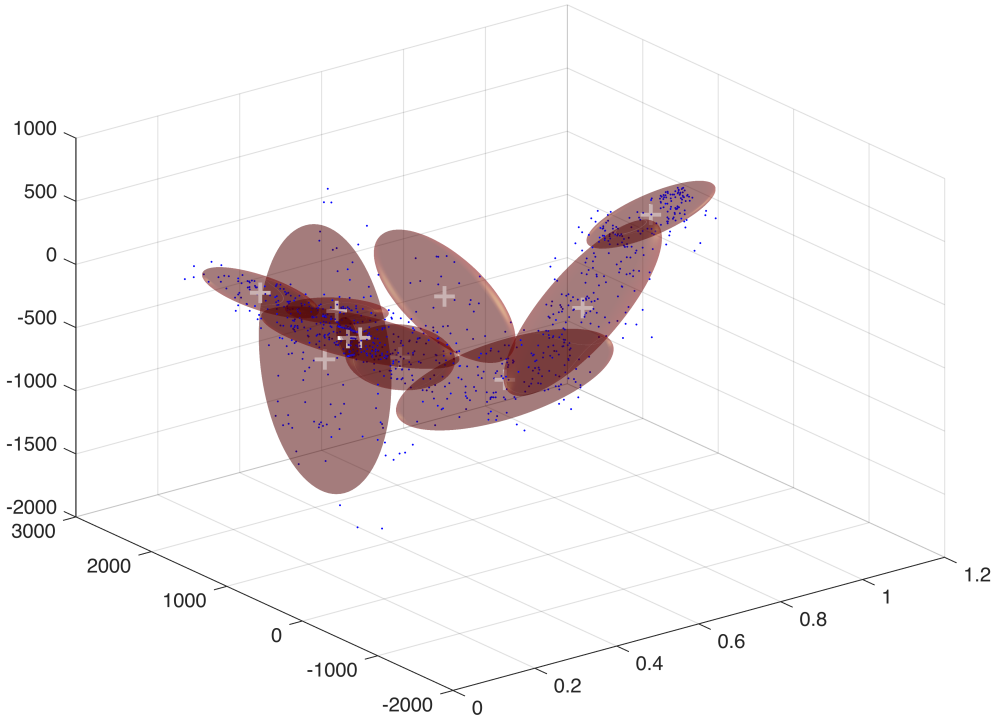
Spatial plot

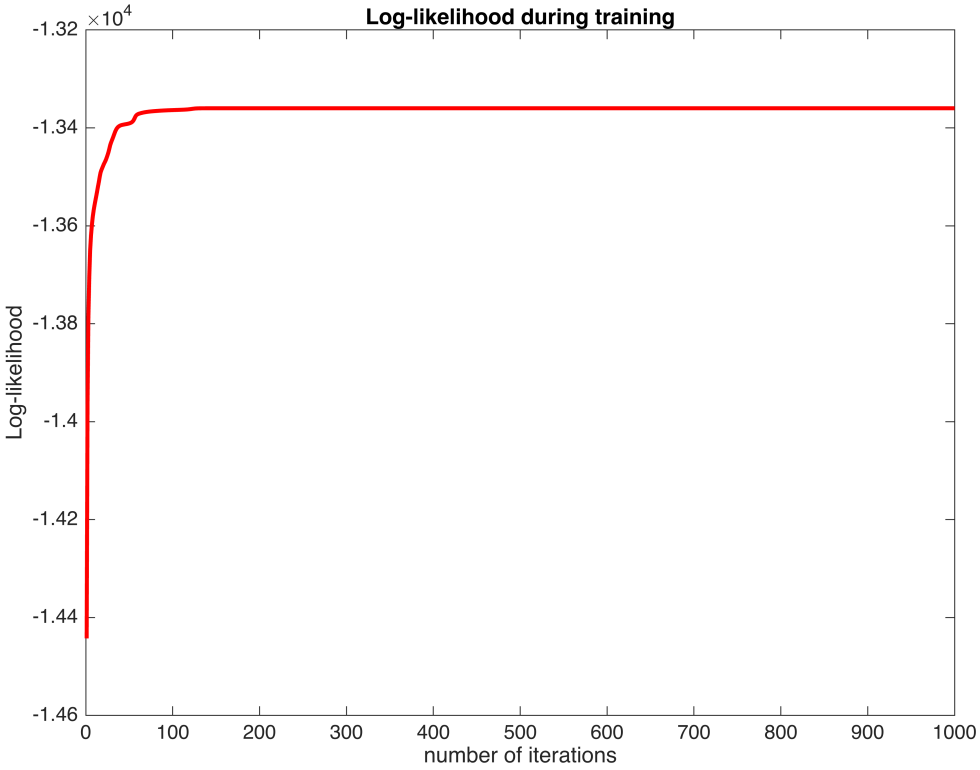
```
figure,
for n = 1:nbDemos
    plot(demos_in{n}.pos(1,:),
demos_in{n}.pos(2,:),'.','markersize',8,'color',[.7 .7 .7]); hold on
end
plot(trajGMR(1,:),trajGMR(2,:),'-','linewidth',3,'color',[0 .7 0]);
xlabel('Postions in the X axis')
ylabel('Postions in the Y axis')
axis equal; axis square; hold off
```

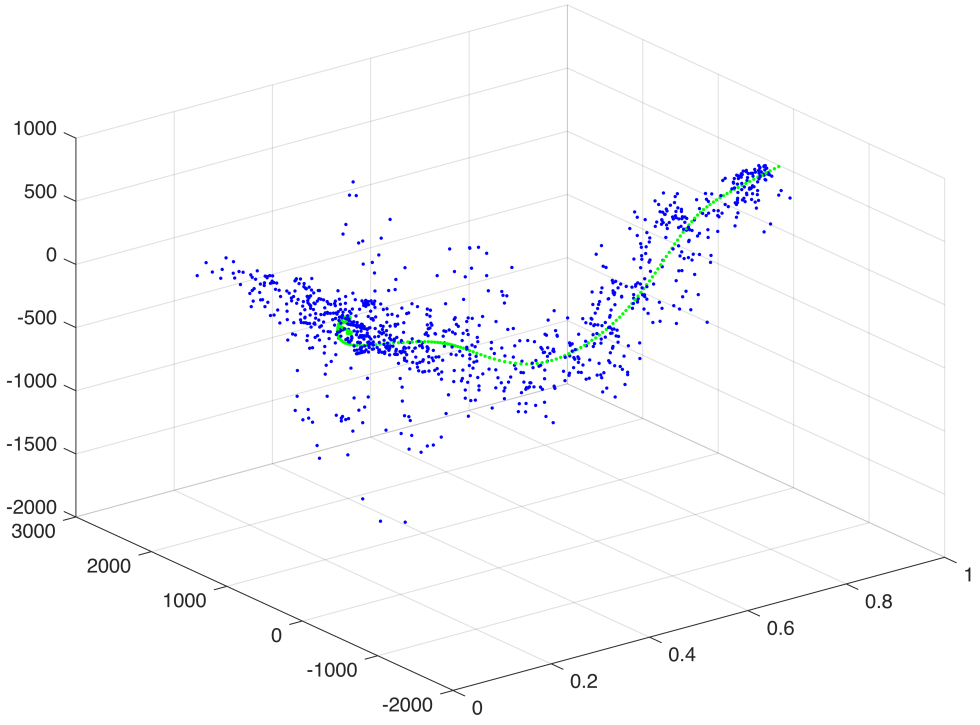


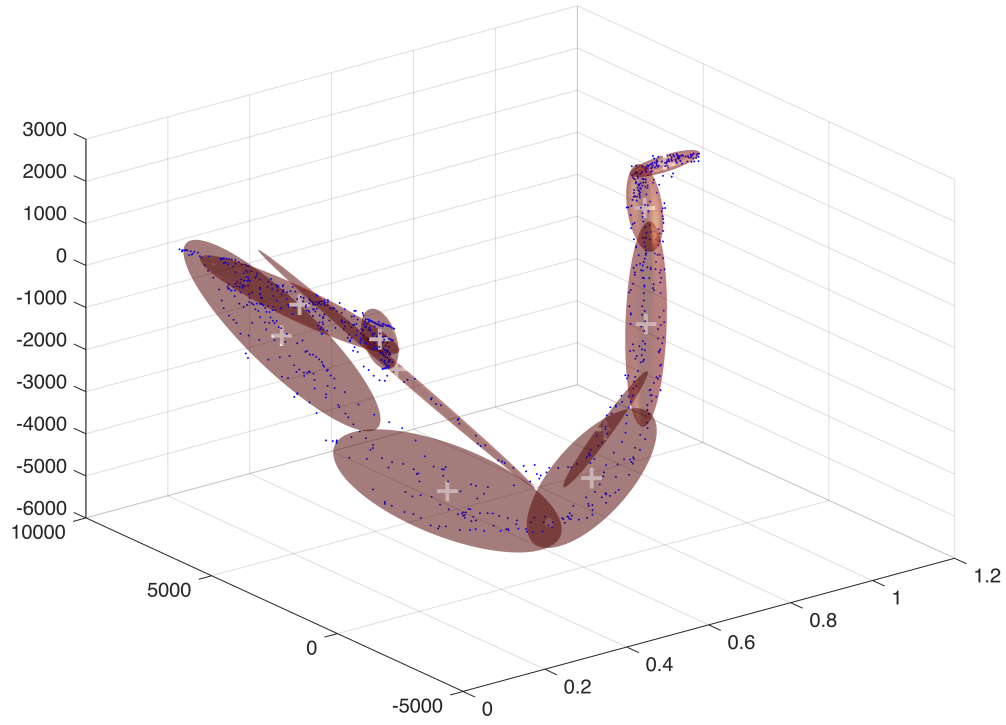

```
% Extra Credit: DMP Parameters change
ext_nbData = 200;                % Length of each demo trajectory
ext_nbDemos = 5;                 % Number of demonstrations for the DMP
training
ext_nStates = 9;                 % Number of activation functions (i.e.,
number of states in the forcing term)
ext_nVar = 3;                    % Number of variables [s,f1,f2] (decay term
and perturbing force)
ext_beta = 50;                   % Stiffness gain (β)
ext_alpha = (2*ext_beta)^.3;         % Damping gain (with ideal underdamped
damping ratio) (α)
ext_decayFactor = 1.1;           % Decay factor (a in τx'=-ax)
dt = 0.01;                    % Duration of time step (τ)
demos = [];
```

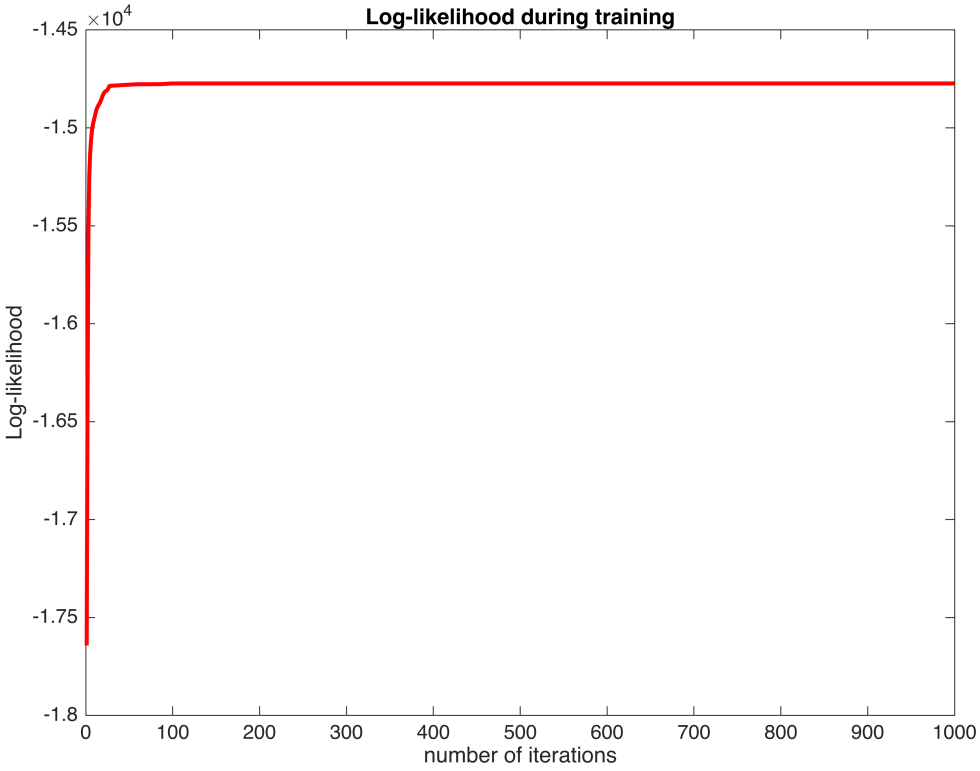
```
load('G.mat');
ext_demos_in = {demos{1:ext_nbDemos}};
ext_trajs = {};
% rmse_GMRs = {};
figure,
for beta = ext_beta:200:1000
    ext_dmp = SkillGeneralisation(ext_nVar, ext_nStates, beta, ext_alpha,
dt);   % Class constructor
    ext_dmp = ext_dmp.canonicalSystemInitialisation(decayFactor, nbData);
    ext_dmp = ext_dmp.trajectort2Forcing(ext_demos_in);
    ext_fGMR = ext_dmp.fittingWithGMR();
    ext_trajGMR = ext_dmp.forcing2Trajectory(ext_fGMR);
    ext_trajs{end+1} = ext_trajGMR;

    rmse_values = zeros(2, ext_nbDemos);
    % for n = 1:ext_nbDemos
    %     rmse = dmp.evalRMSE(ext_demos_in{n}.pos, ext_trajGMR);
    %     rmse_values(1, n) = rmse(1);
    %     rmse_values(2, n) = rmse(2);
    % end
    % rmse_GMRs{end+1} = [mean(rmse_values(1, :)),mean(rmse_values(2, :))];
end
```

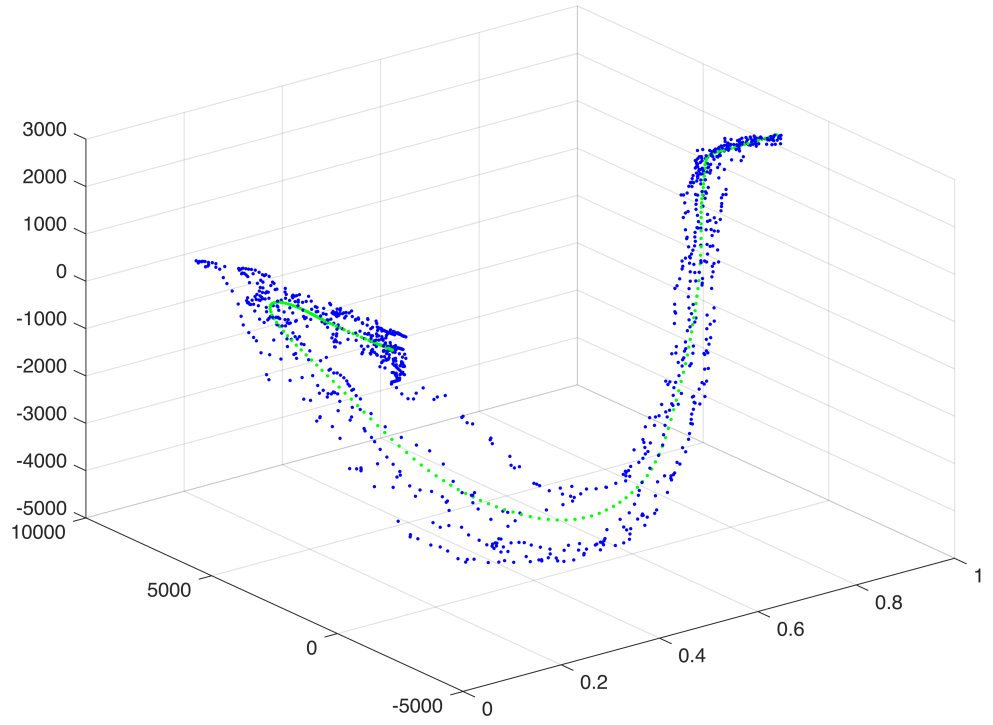


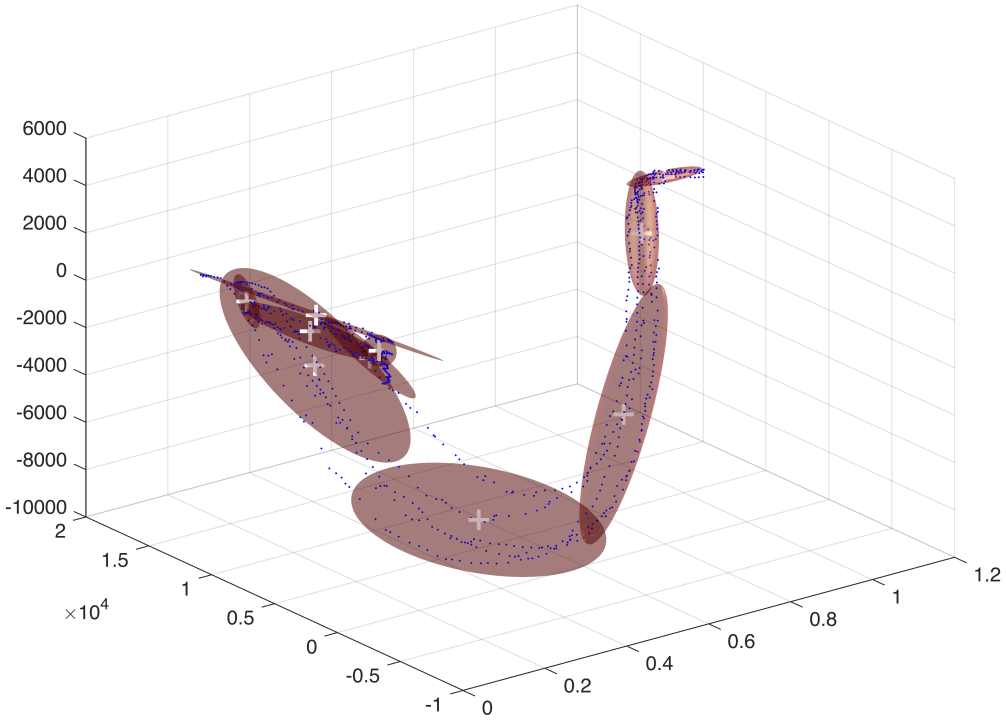

Log-likelihood during training
×10^4
-1.32
-1.34
-1.36
-1.38
-1.4
-1.42
-1.44
-1.46
Log-likelihood
0
100
200
300
400
500
600
700
800
900
1000
number of iterations

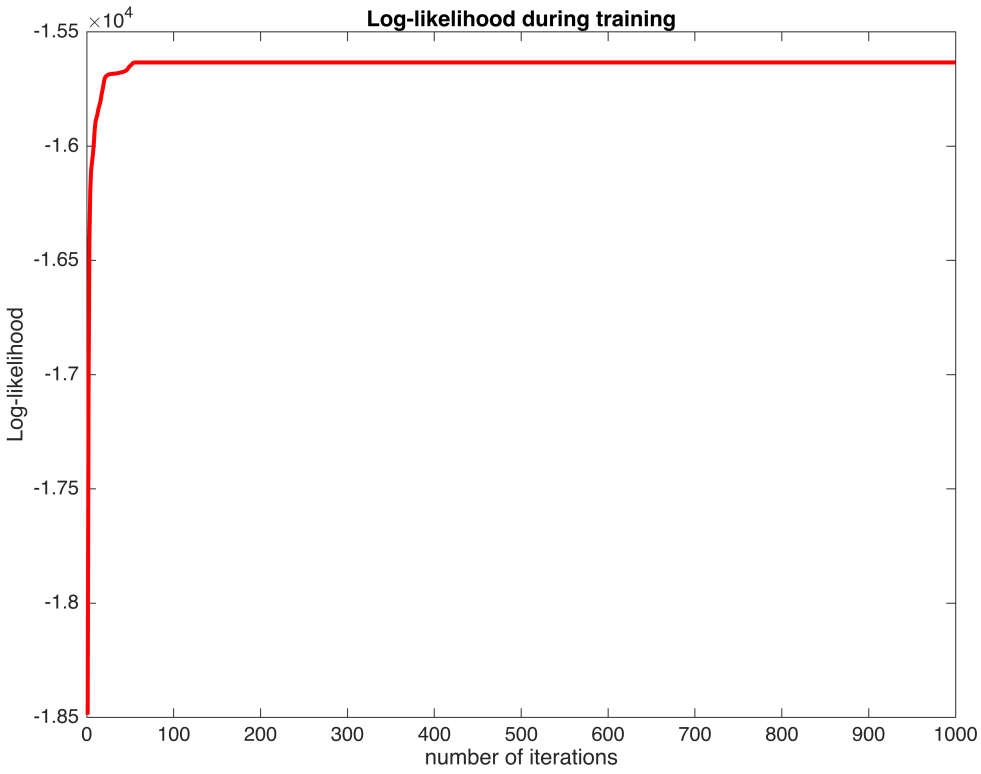

1000
500
0
-500
-1000
-1500
-2000
3000
2000
1000
0
-1000
-2000
0
0.2
0.4
0.6
0.8
1

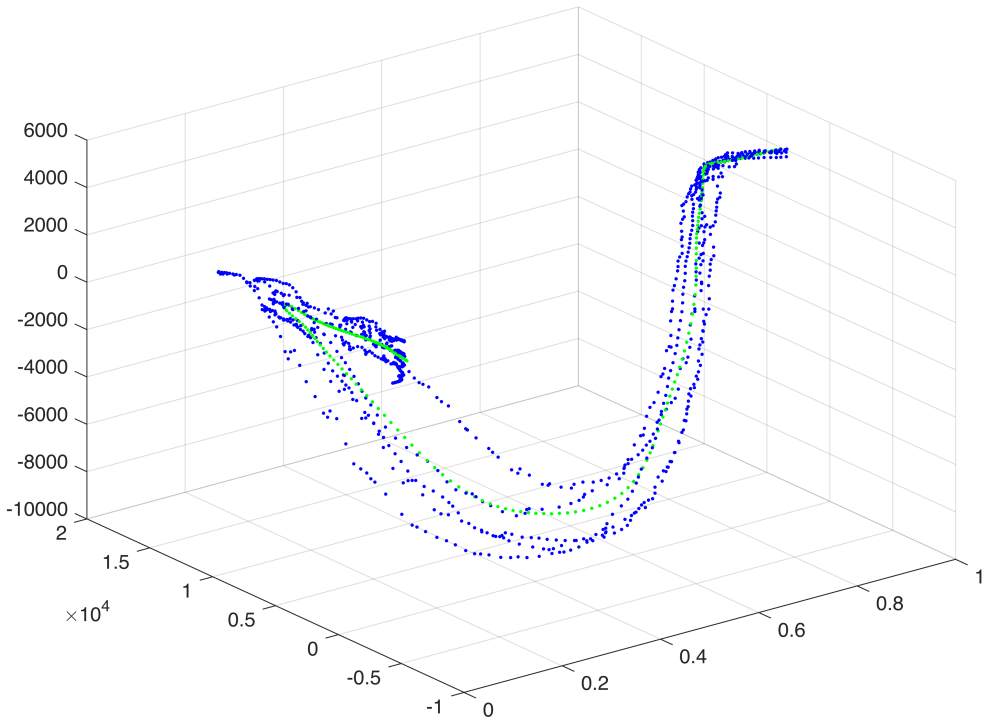

3000
2000
1000
0
-1000
-2000
-3000
-4000
-5000
-6000
10000
5000
0
-5000
0
0.2
0.4
0.6
0.8
1
1.2

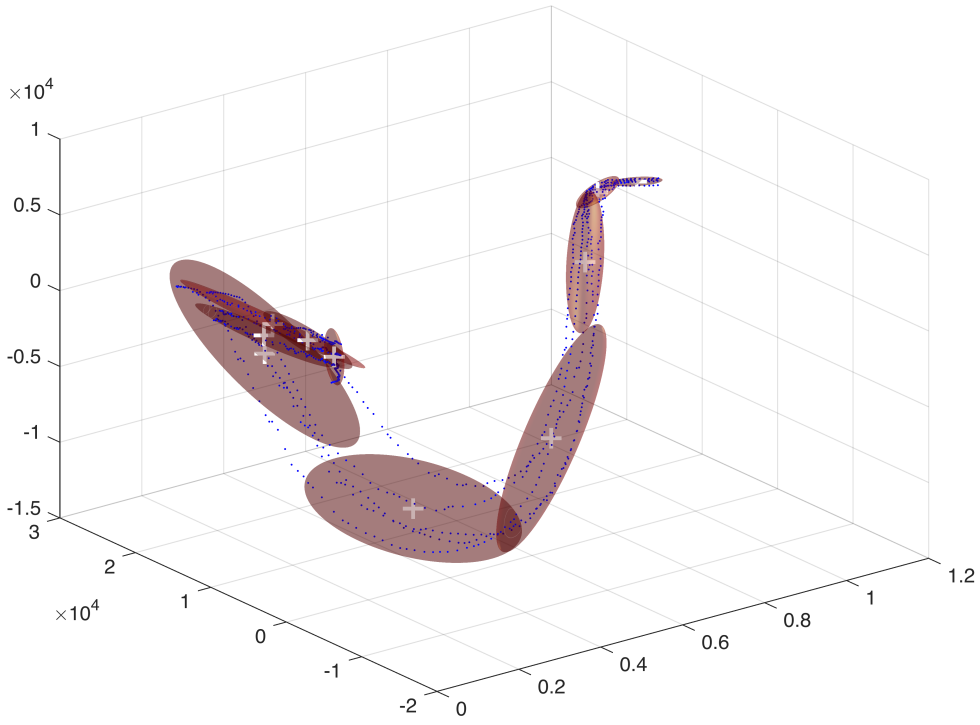

Log-likelihood during training
×10^4
-1.45
-1.5
-1.55
-1.6
-1.65
-1.7
-1.75
-1.8
Log-likelihood
0
100
200
300
400
500
600
700
800
900
1000
number of iterations

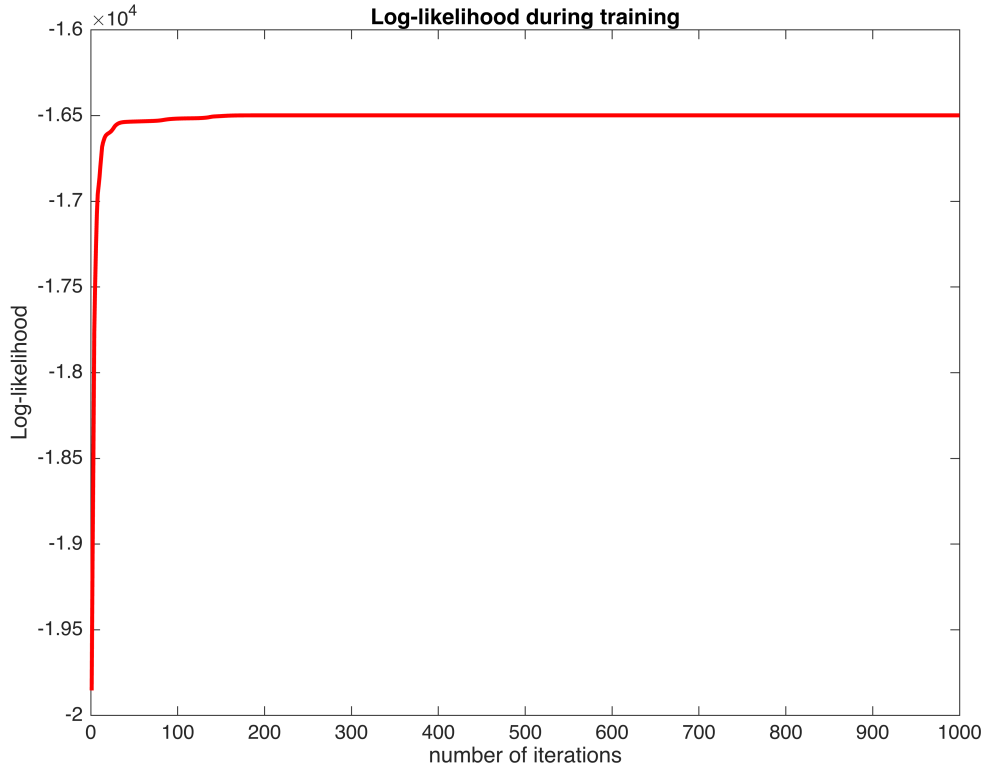

3000
2000
1000
0
-1000
-2000
-3000
-4000
-5000
10000
5000
0
-5000
0
0.2
0.4
0.6
0.8
1

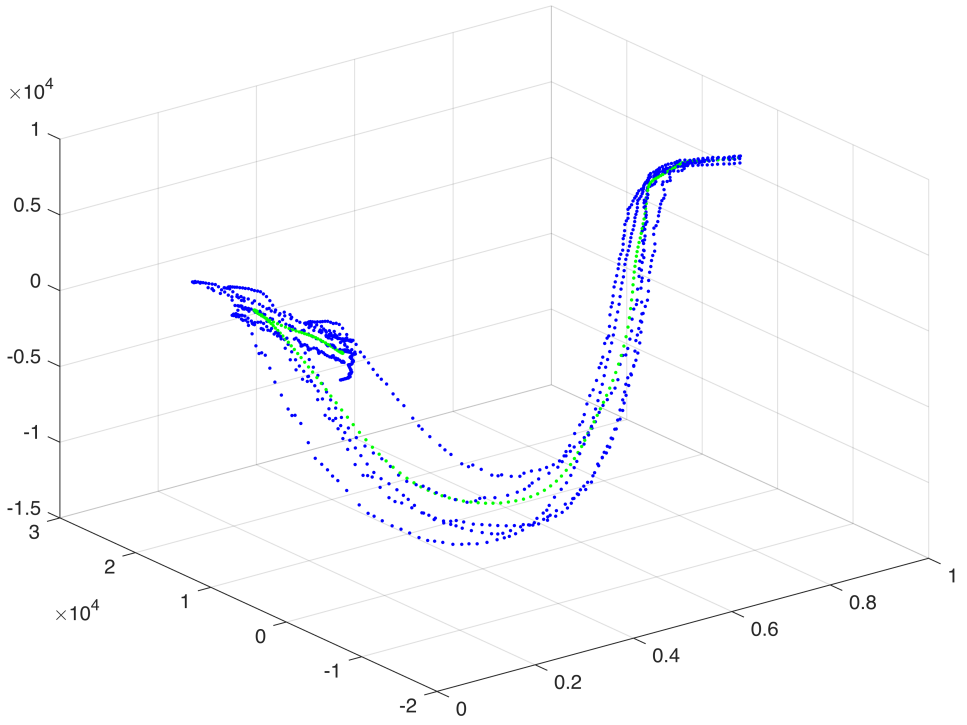

6000
4000
2000
0
-2000
-4000
-6000
-8000
-10000
2
1.5
1
0.5
0
-0.5
-1
×10^4
0
0.2
0.4
0.6
0.8
1
1.2

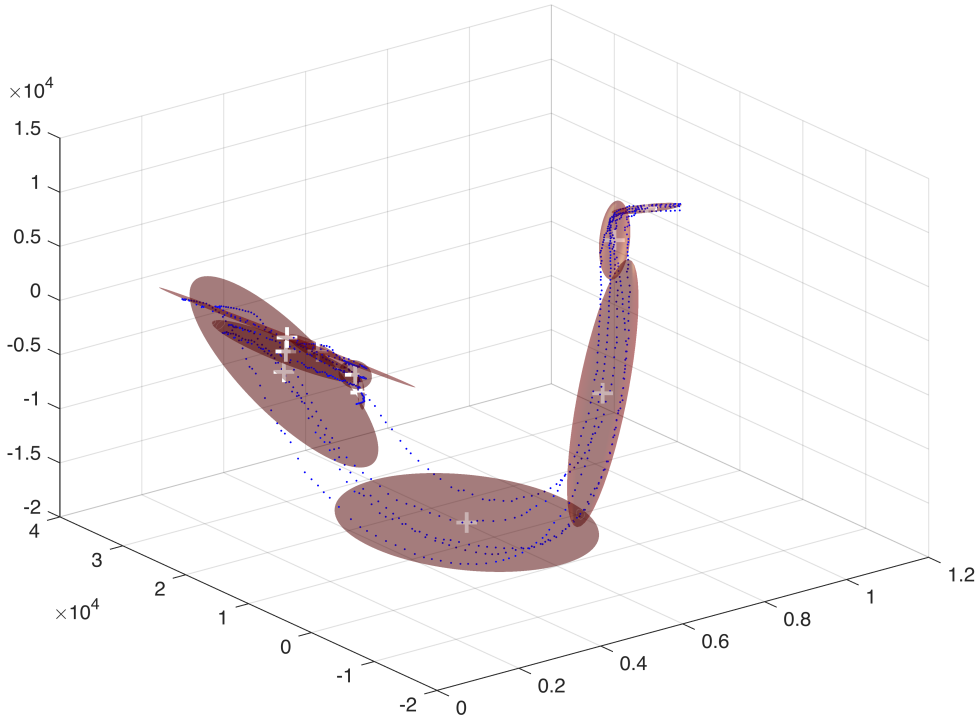

Log-likelihood during training
×10^4
-1.55
-1.6
-1.65
-1.7
-1.75
-1.8
-1.85
Log-likelihood
0
100
200
300
400
500
600
700
800
900
1000
number of iterations

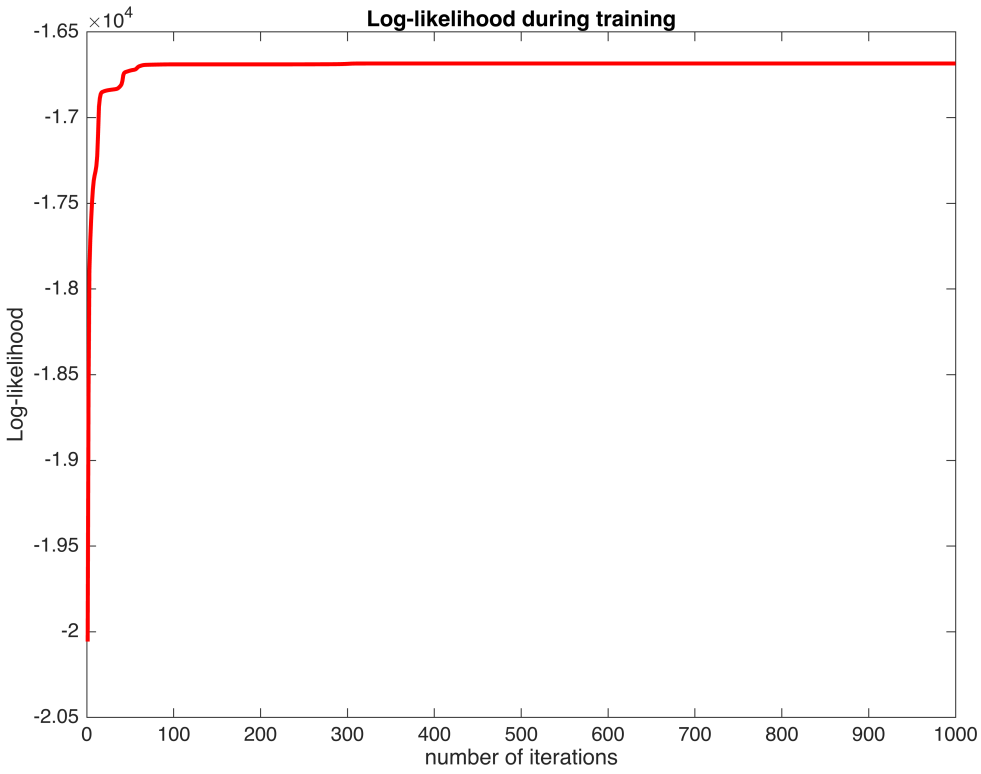

6000
4000
2000
0
-2000
-4000
-6000
-8000
-10000
2
1.5
1
0.5
0
-0.5
-1
×10^4
0
0.2
0.4
0.6
0.8
1

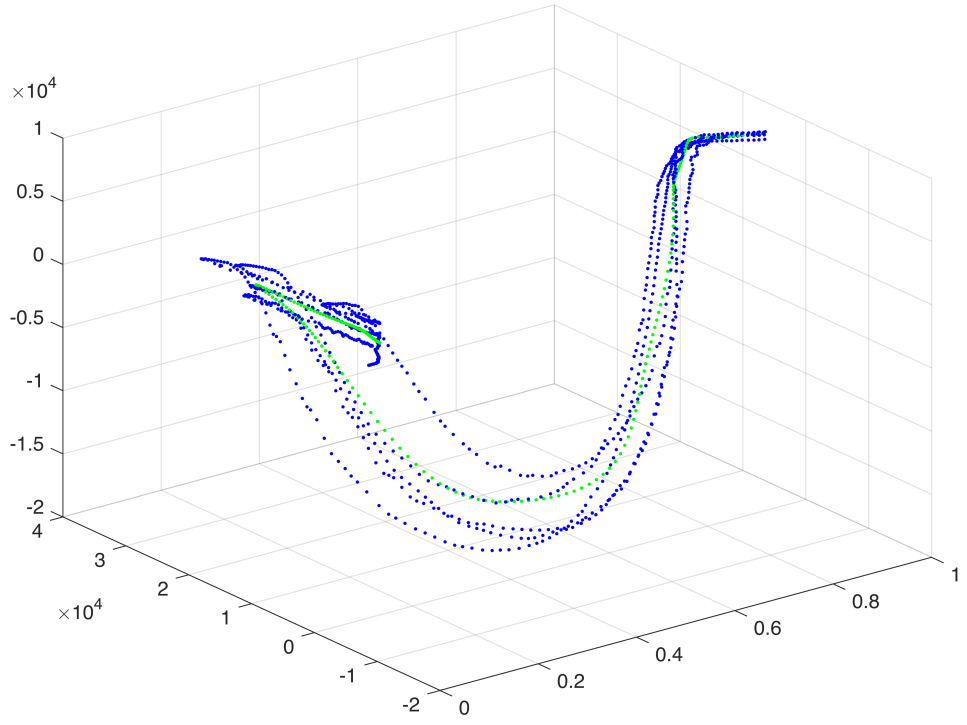

×10^4
1
0.5
0
-0.5
-1
-1.5
3
2
×10^4
1
0
-1
-2
0
0.2
0.4
0.6
0.8
1
1.2


Log-likelihood during training
×10^4
-1.6
-1.65
-1.7
-1.75
-1.8
-1.85
-1.9
-1.95
-2
Log-likelihood
0
100
200
300
400
500
600
700
800
900
1000
number of iterations


×10^4
1
0.5
0
-0.5
-1
-1.5
3
2
1
0
-1
-2
×10^4
0
0.2
0.4
0.6
0.8
1

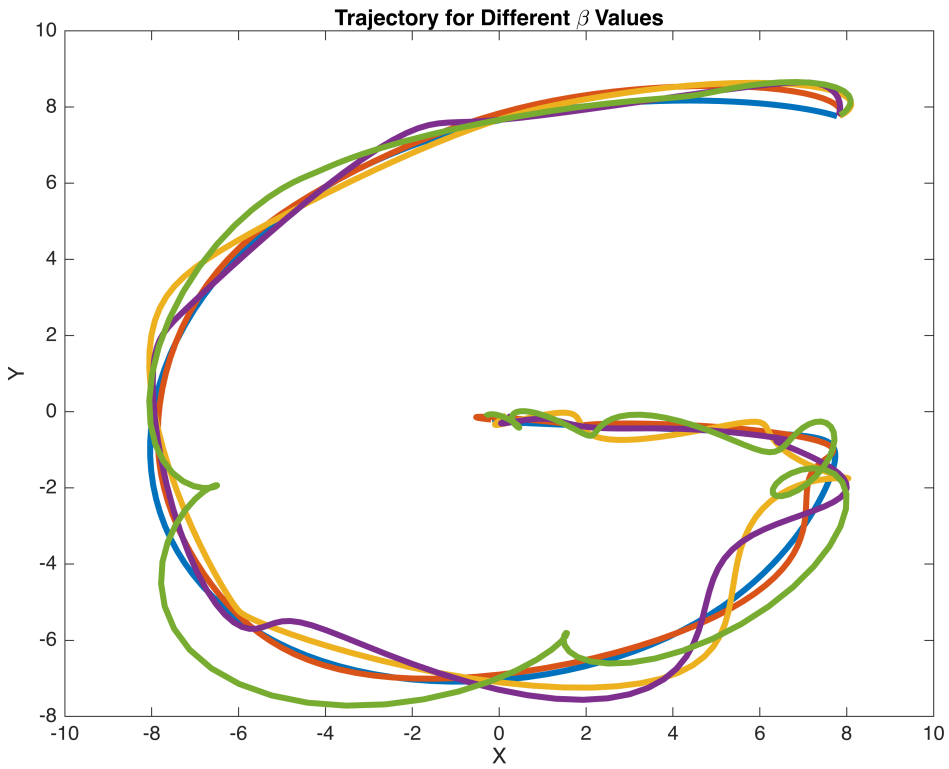

×10^4
1.5
1
0.5
0
-0.5
-1
-1.5
-2
4
3
2
1
0
-1
-2
×10^4
0
0.2
0.4
0.6
0.8
1
1.2






```
figure,
for i = 1:length(ext_trajs)
```

```
    plot(ext_trajs{i}(1, :), ext_trajs{i}(2,:), '-', 'LineWidth', 3); hold
on;
end

xlabel('X');
ylabel('Y');
title('Trajectory for Different \beta Values');
hold off;
```



The figure above shows that increasing the stiffness term (spring constant) in the DMP equation may cause the system to become **underdamped**, resulting in **oscillatory behavior**.

Part 3 (5 marks):

- Once the new trajectory has been learned and generalized, save the data in a .mat format to the workspace

Useful materials for this Part include:

- The save function documentation: Save variables from workspace to file - MATLAB save - MathWorks United Kingdom

```
% Add code to generate and save your results here. Please only save the
% relevant variables, not the entire workspace
trajGMR_simulink = trajGMR;
```

```
save("GMR_G.mat", "trajGMR_simulink")
```

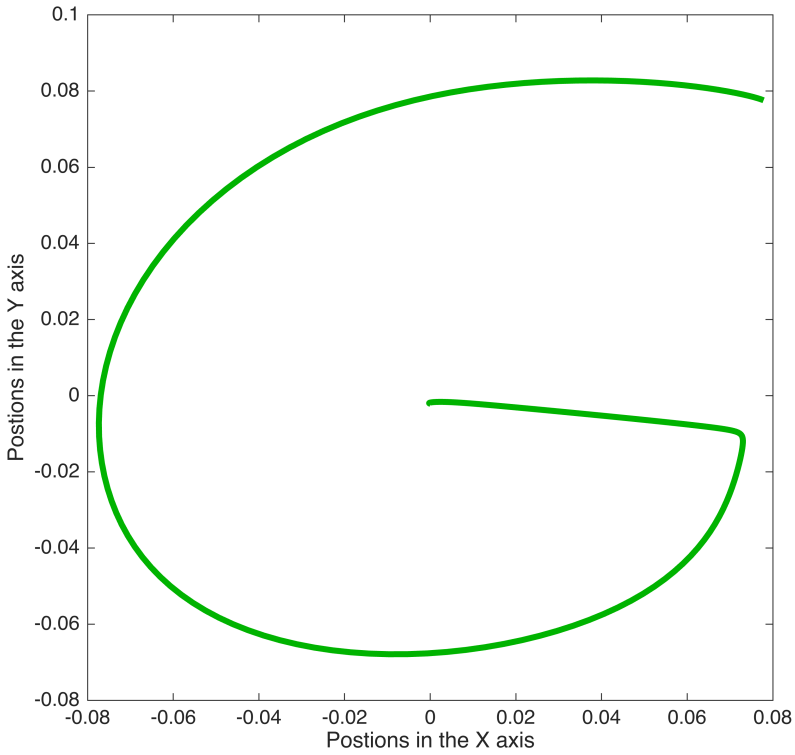
Part 4 (15 marks):

- During teleoperation simulation, load the saved data as inputs for the teleoperation system to facilitate further simulation.
- Please complete simulations and make a screenshot of the simulation results with the scopes to illustrate the simulation results
- Be sure to save your data from Simulink with the filename 'SimOut.mat'

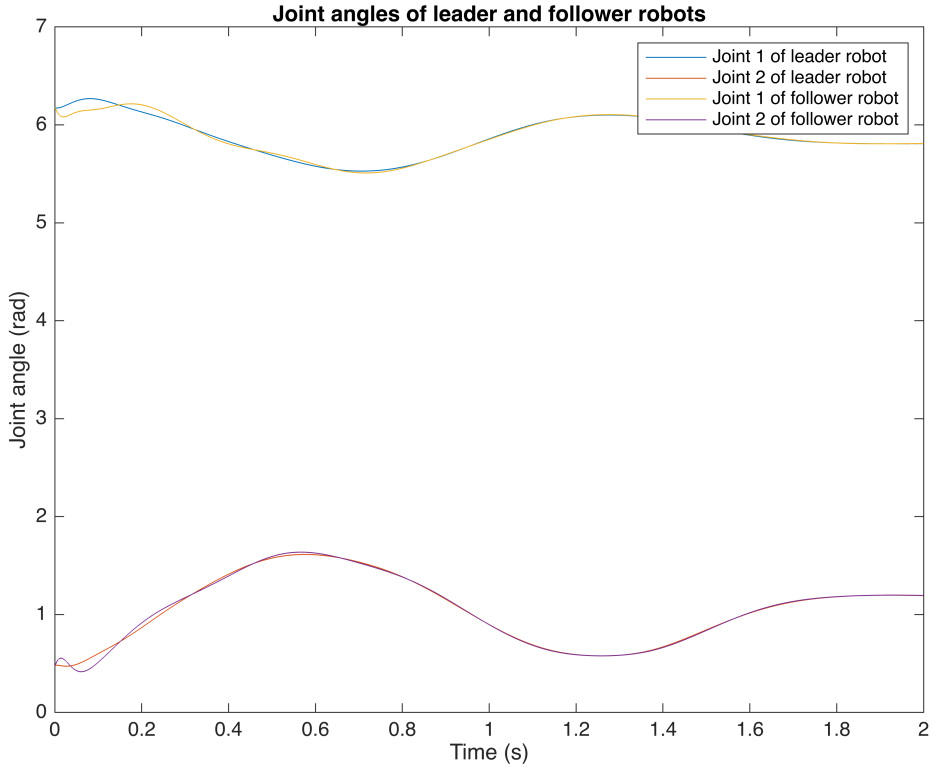
Useful materials for this Part include:

- The lecture material on Robot Control

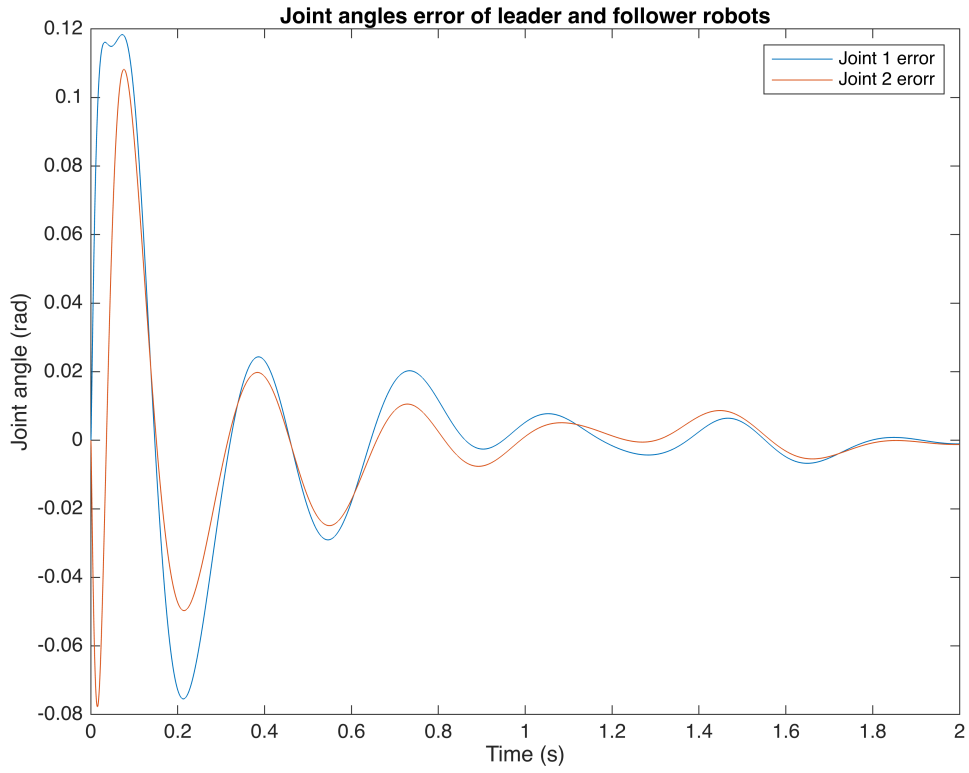
```
% Add any external code to run the teleoperation simulation here, if
required
% Ensure your Simulink model loads and is visible for review
load("GMR_G.mat")
[pos, vel, acc] = dmp.generateHighOrderTerms(trajGMR_simulink);

% Scale down the trajectories to match the Simulink model
pos = pos / 100;

% This figure show the trajectory that is an inputs for the teleoperation
% system
figure;
plot(pos(1, :),pos(2, :),'-','linewidth',3,'color',[0 .7 0]);
xlabel('Postions in the X axis')
ylabel('Postions in the Y axis')
axis equal; axis square; hold off
```



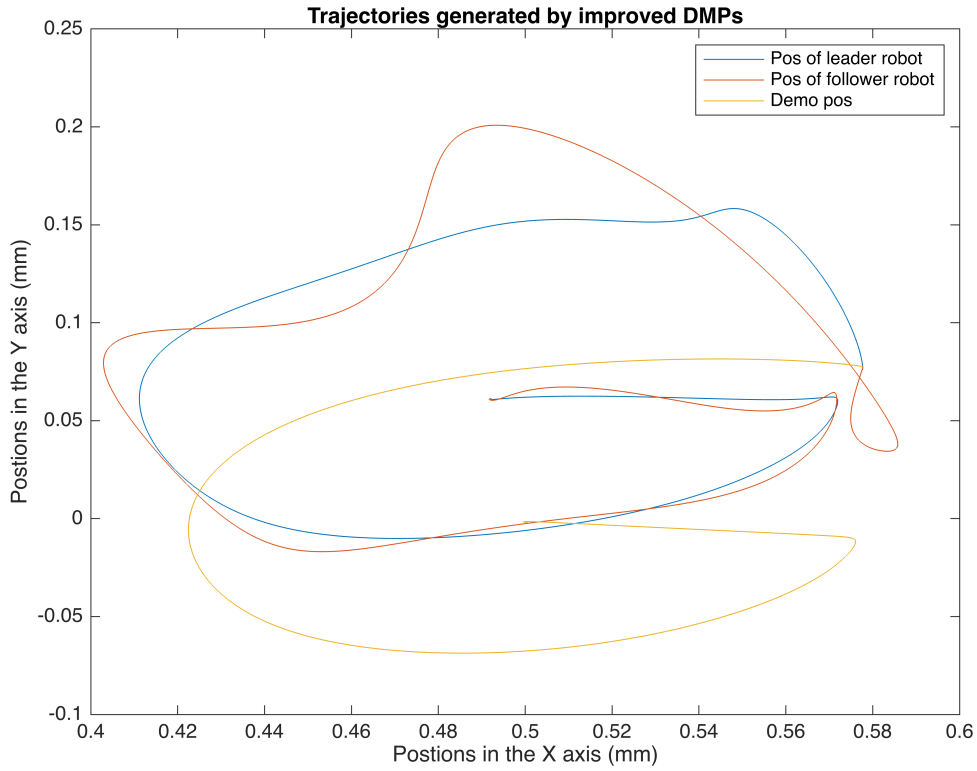

```
G = [pos(1, :)', pos(2, :)'];
G_time = linspace(0, 2, size(pos, 2))';
ts1 = timeseries(G, G_time);

% Plot joint between leader and follower
load("SimOut.mat")
Time = out.tout';
figure,
Joint1 = out.joint.data(:,1);
Joint2 = out.joint.data(:,2);
Joint3 = out.joint.data(:,3);
Joint4 = out.joint.data(:,4);
plot(Time,Joint1); hold on; plot(Time,Joint2); hold on
plot(Time,Joint3); hold on; plot(Time,Joint4);
title("Joint angles of leader and follower robots")
legend('Joint 1 of leader robot','Joint 2 of leader robot',...
    'Joint 1 of follower robot','Joint 2 of follower robot')
xlabel('Time (s)');ylabel('Joint angle (rad)')
```



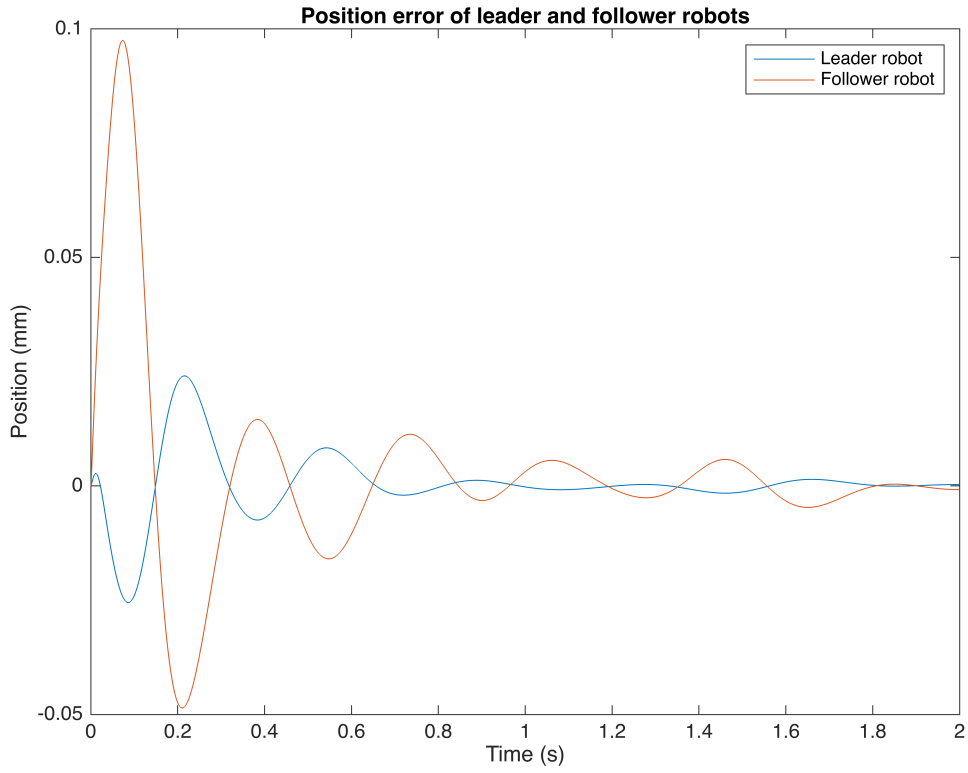

```
% Plot joint error
figure,
Joint1_error = out.joint_error.data(:,1);
Joint2_error = out.joint_error.data(:,2);
plot(Time,Joint1_error); hold on; plot(Time,Joint2_error);
title("Joint angles error of leader and follower robots")
legend('Joint 1 error','Joint 2 erorr')
xlabel('Time (s)');ylabel('Joint angle (rad)')
```



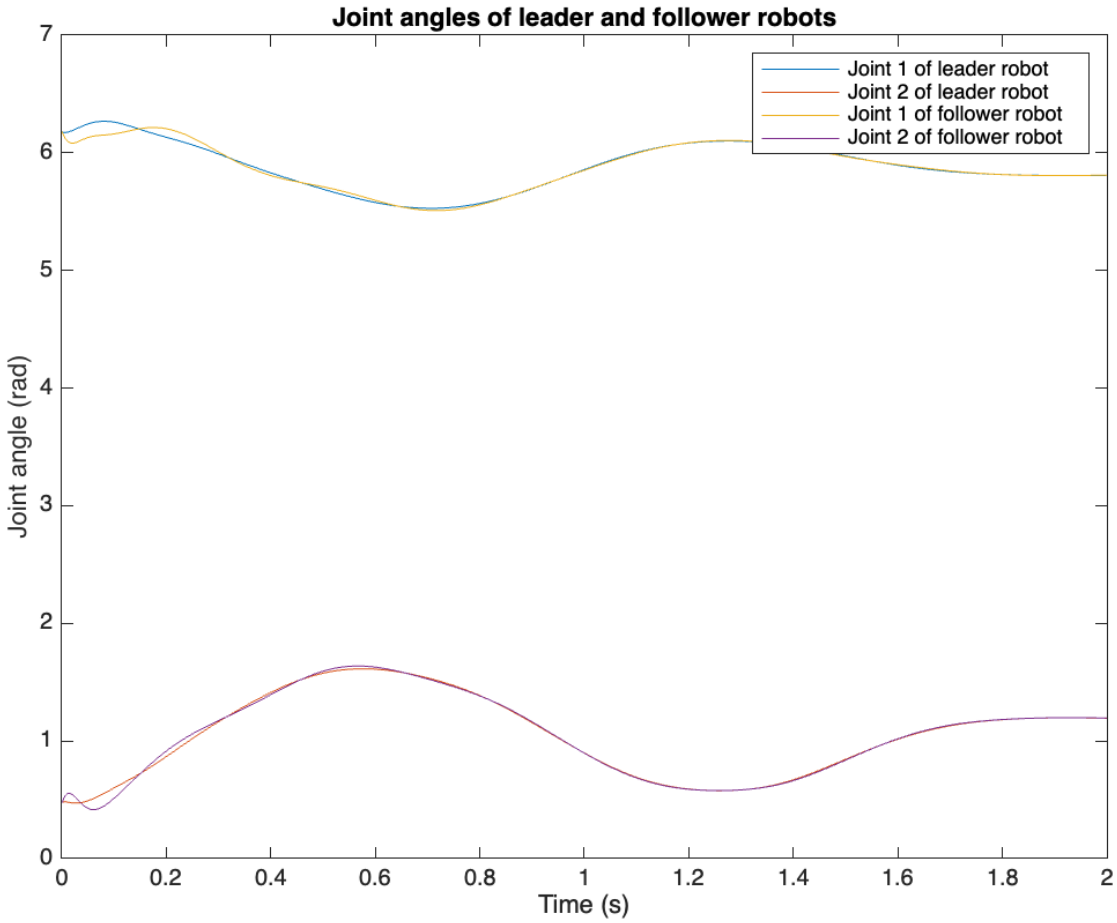

```
% Plot trajectories of demos, leader robot, and follower robot
figure,
plot(out.pos.data(:,1),out.pos.data(:,2)); hold on;
plot(out.pos.data(:,3),out.pos.data(:,4)); hold on
plot(out.pos.data(:,5),out.pos.data(:,6));
legend('Pos of leader robot', 'Pos of follower robot','Demo pos')
title("Trajectories generated by improved DMPs")
xlabel('Postions in the X axis (mm)')
ylabel('Postions in the Y axis (mm)')
```

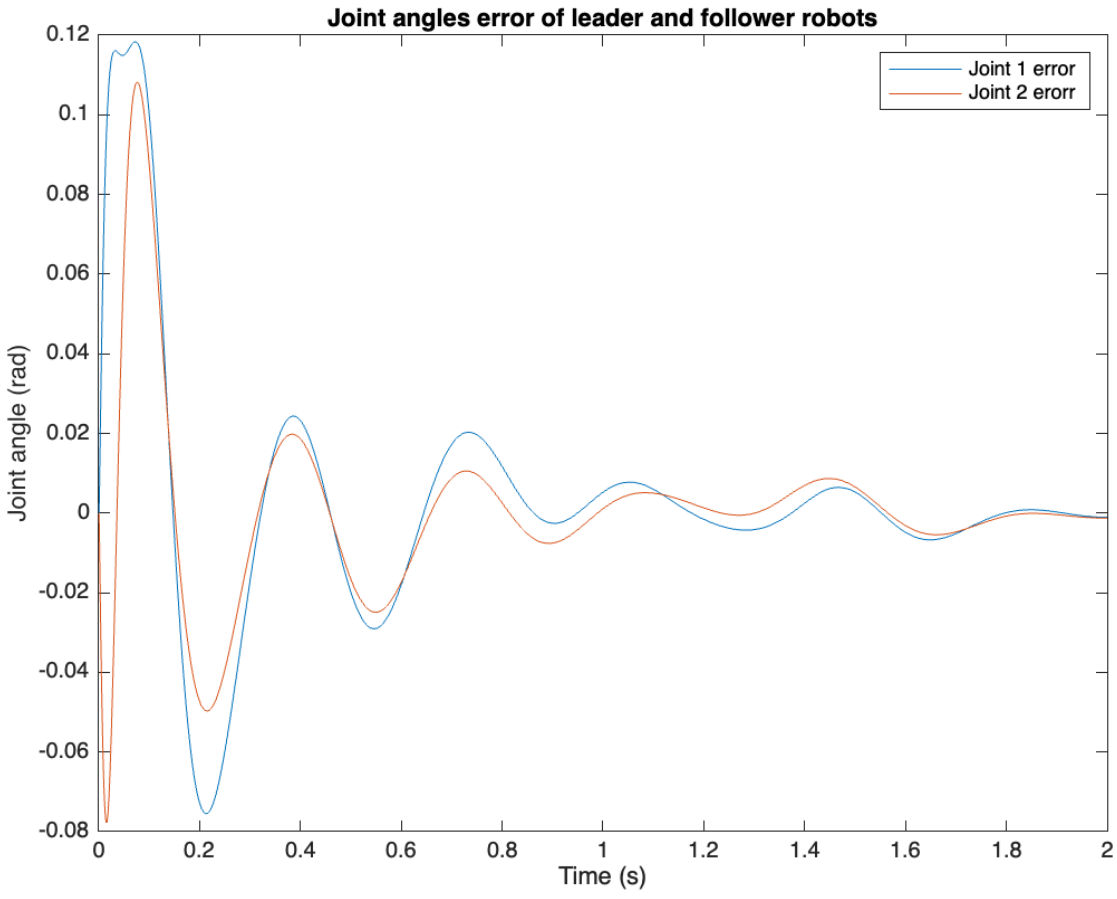


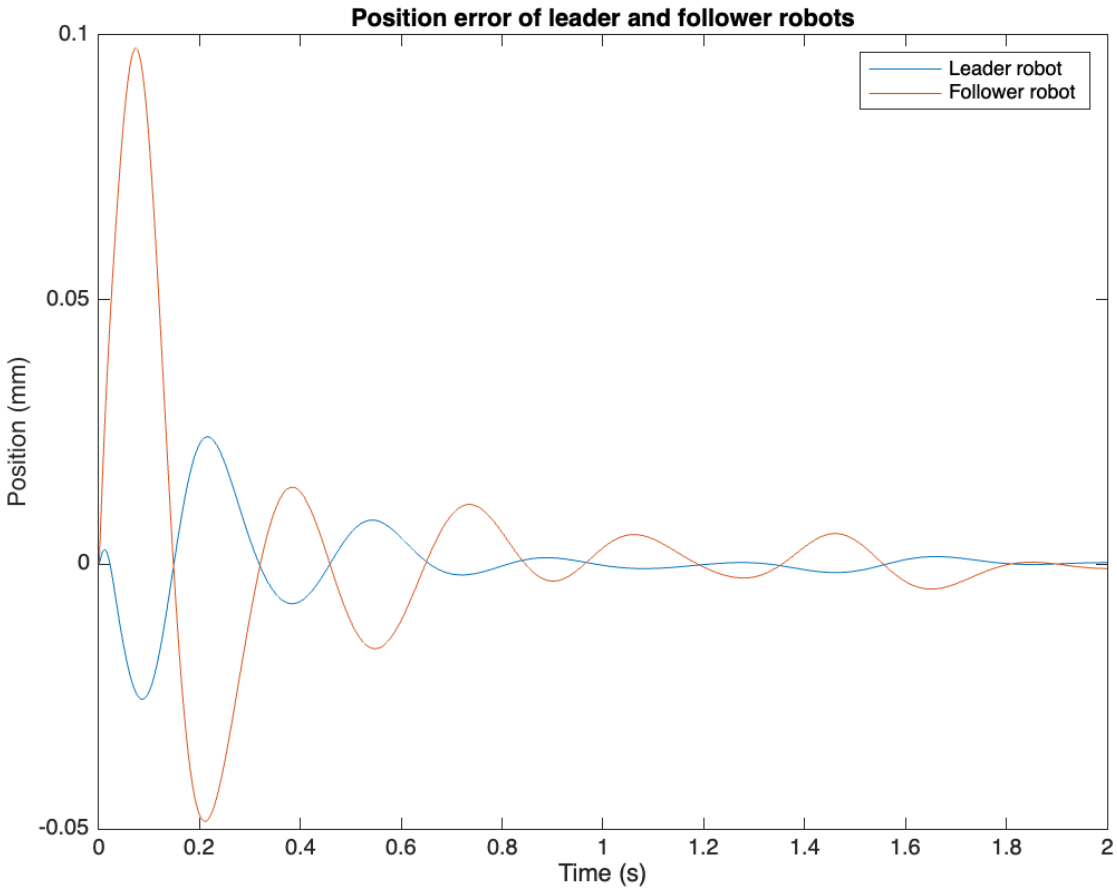

```
% Plot position error
figure,
leader_pos_err = out.pos_err.data(:,1);
follower_pos_err = out.pos_err.data(:,2);
plot(Time,leader_pos_err); hold on; plot(Time,follower_pos_err);
title("Position error of leader and follower robots")
legend('Leader robot','Follower robot')
xlabel('Time (s)');ylabel('Position (mm)')
```



Add screenshots supporting your results here


Joint angles of leader and follower robots
Joint 1 of leader robot
Joint 2 of leader robot
Joint 1 of follower robot
Joint 2 of follower robot
Joint angle (rad)
Time (s)
0
1
2
3
4
5
6
7
0.2
0.4
0.6
0.8
1.2
1.4
1.6
1.8


Joint angles error of leader and follower robots
Joint 1 error
Joint 2 erorr
Joint angle (rad)
Time (s)
0.12
0.1
0.08
0.06
0.04
0.02
0
-0.02
-0.04
-0.06
-0.08
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2



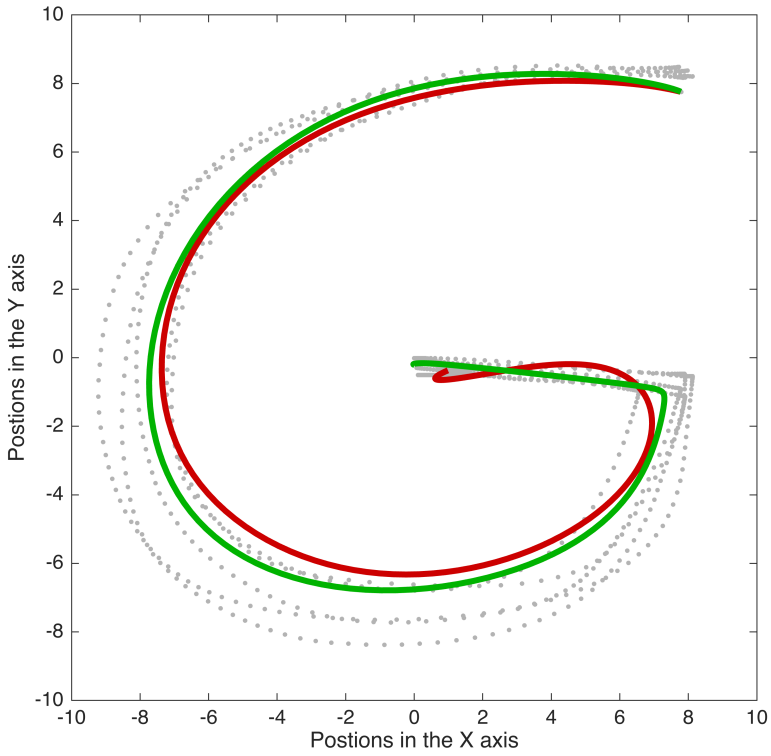
Task 3 (15 marks)

- Evaluate the proposed enhanced Dynamic Movement Primitives method by comparing the differences between the original Dynamic Movement Primitives and the enhanced method in Cartesian space coordinates.
- The successful results should demonstrate comparative errors between two different methods.
- Complete the comparison, writing (static) functions in the `RobotLearning.m` or your code here to quantify and compare your results.
- Describe and analyse these results.

Below is a loop to generate an appropriate figure to view your results. Use this as a starting point to demonstrate the differences between original and enhanced DMPs (which you explored in Tasks 1 and 2 respectively).

```
figure,
for n = 1:nbDemos
    plot(demos_in{n}.pos(1,:),
demos_in{n}.pos(2,:),'.','markersize',8,'color',[.7 .7 .7]); hold on
end
plot(trajWLR(1,:),trajWLR(2,:),'-','linewidth',3,'color',[.8 0 0]);
```

```
plot(trajGMR(1,:),trajGMR(2,:),'-','linewidth',3,'color',[0 .7 0]);
xlabel('Postions in the X axis')
ylabel('Postions in the Y axis')
axis equal; axis square; hold off
```




```
nbTotalDemos = length(demos) - nbDemos;
demos_test = {demos{nbDemos:end}};

err = struct();
err.wlr_train = zeros(3, nbDemos);
err.wlr_test = zeros(3, nbTotalDemos);
err.gmr_train = zeros(3, nbDemos);
err.gmr_test = zeros(3, nbTotalDemos);

% Accuracy evaluation euclidean distance RMSE from training data
for n = 1:nbDemos
    wlr = dmp.evalRMSE(demos_in{n}.pos, trajWLR);
    err.wlr_train(1, n) = wlr(1);
    err.wlr_train(2, n) = wlr(2);
    err.wlr_train(3, n) = sqrt(wlr(1)^2 + wlr(2)^2);

    gmr = dmp.evalRMSE(demos_in{n}.pos, trajGMR);
    err.gmr_train(1, n) = gmr(1);
    err.gmr_train(2, n) = gmr(2);
    err.gmr_train(3, n) = sqrt(gmr(1)^2 + gmr(2)^2);
end
```

```
% Accuracy evaluation euclidean distance RMSE from test data
for n = 1:nbTotalDemos
    wlr = dmp.evalRMSE(demos_test{n}.pos, trajWLR);
    err.wlr_test(1, n) = wlr(1);
    err.wlr_test(2, n) = wlr(2);
    err.wlr_test(3, n) = sqrt(wlr(1)^2 + wlr(2)^2);

    gmr = dmp.evalRMSE(demos_test{n}.pos, trajGMR);
    err.gmr_test(1, n) = gmr(1);
    err.gmr_test(2, n) = gmr(2);
    err.gmr_test(3, n) = sqrt(gmr(1)^2 + gmr(2)^2);
end

Method = ["Original";"Improved"];
X = [[mean(err.wlr_train(1, :)), std(err.wlr_train(1, :))];
[mean(err.gmr_train(1, :)), std(err.gmr_train(1, :))]];
Y = [[mean(err.wlr_train(2, :)), std(err.wlr_train(2, :))];
[mean(err.gmr_train(2, :)), std(err.gmr_train(2, :))]];
Euclidean = [[mean(err.wlr_train(3, :)), std(err.wlr_train(3, :))];
[mean(err.gmr_train(3, :)), std(err.gmr_train(3, :))]];

trainning_error = table(Method, ...
    X(:,1), X(:,2), ...
    Y(:,1), Y(:,2), ...
    Euclidean(:,1), Euclidean(:,2), ...
    'VariableNames', {'Trainning error', 'X_mean', 'X_std', 'Y_mean',
'Y_std', 'Euclidean_mean', 'Euclidean_std'})
```

```
trainning_error = 2×7 table
                                                                    ...
```

	Trainning error	X_mean	X_std	Y_mean	Y_std	Euclidean_mean
1	"Original"	1.6678	0.5466	1.1673	0.3745	2.0405
2	"Improved"	1.3433	0.6072	1.0861	0.4201	1.7364

```
X = [[mean(err.wlr_test(1, :)), std(err.wlr_test(1, :))];
[mean(err.gmr_test(1, :)), std(err.gmr_test(1, :))]];
Y = [[mean(err.wlr_test(2, :)), std(err.wlr_test(2, :))];
[mean(err.gmr_test(2, :)), std(err.gmr_test(2, :))]];
Euclidean = [[mean(err.wlr_test(3, :)), std(err.wlr_test(3, :))];
[mean(err.gmr_test(3, :)), std(err.gmr_test(3, :))]];

test_error = table(Method, ...
    X(:,1), X(:,2), ...
    Y(:,1), Y(:,2), ...
    Euclidean(:,1), Euclidean(:,2), ...
    'VariableNames', {'Test error', 'X_mean', 'X_std', 'Y_mean', 'Y_std',
'Euclidean_mean', 'Euclidean_std'})
```

```
test_error = 2×7 table
                                                                    ...
```

	Test error	X_mean	X_std	Y_mean	Y_std	Euclidean_mean
1	"Original"	1.9041	0.5935	1.3171	0.4642	2.3351
2	"Improved"	1.5056	0.5938	1.2382	0.5163	1.9629

Describe your results below, comparing the trajectories generated by original vs improved DMPs (max. 500 words):

Important note: improved DMPs error will subject to change in value slightly every time the code is re-run due to random inital parameters

In this task, we use **x_error**, **y_error**, and **euclidean distance error** from each trajectories to represent the performance of orignal and improved DMPs. Root Mean Square Error (RMSE) is selected to evaluate the accuracy, robustness, and generalisation of each regression model. From the 13 available demonstration datasets, 5 are used for training and 8 for testing. We refer the RMSE from the training dataset as the **trainning error**, and the RMSE from the test dataset as the **test error**. The **mean error** from Cartesian space coordinates (x,y, and euclidean distance) represents the accuracy of the model. The **standard deviation error** from Cartesian space coordinates represents the robustness of the model. The **test error** represents the generalisation ability of each model to unseen data.

According the results shown in the *traning_error* table, the improved DMPs outperfrom the original DMPs in terms of Cartesian accuracy. The improved DMPs achieve lower means of X, Y, and Euclidean trainning error of **1.3289 mm, 1.0874 mm, and 1.7280 mm**, compared to **1.6678 mm, 1.1673 mm, and 2.0405 mm** for the original DMPs. However, the Euclidean standard deviation of the trainning error for the original DMPs is **0.6441 mm,** slightly lower than that of the improved DMPs at **0.7165 mm**. This suggests that the original DMPs is more robust accross different trainning demonstrations.

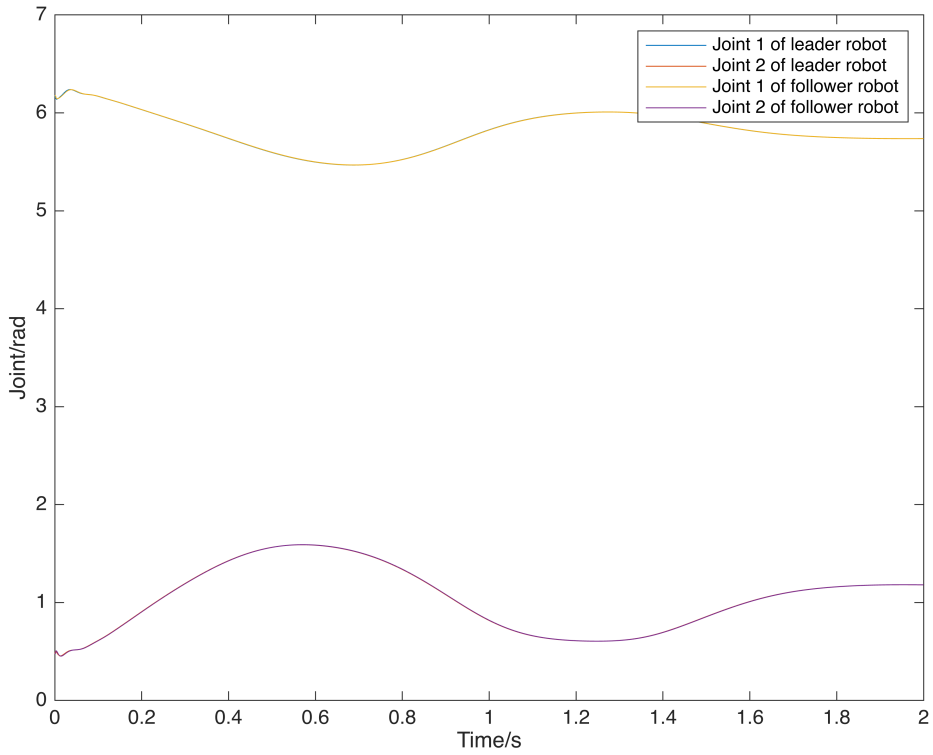
When evaluating generalisation performance on the test set (*test_error* table), the improved DMPs again demonstrate better performance, achieving a lower Euclidean test error of **1.9646 mm** compared to **2.3351 mm** for the original DMPs. Similary, RMSE of x and y coordinates also share the same trend of Euclidean error. These show that the improved DMPs is more genealisae to unseen demonstrations than original DMPs. In terms of robustness, although the improved DMPs are more accurate on average, their slightly higher standard deviation in test error indicates greater variability in performance. Conversely, the original DMPs, while less accurate, show more stable behaviour across the test samples.

Task 4 (15 marks):

- Save the outputs of the robot learning task above, and load them into the teleoperation model in Simulink
- These outputs should become the inputs on the leader controller side to guide the movement of the follower.
- Design PD or other controllers for both the leader and follower; a good implementation will cause the follower's motions to mirror those of the leader.

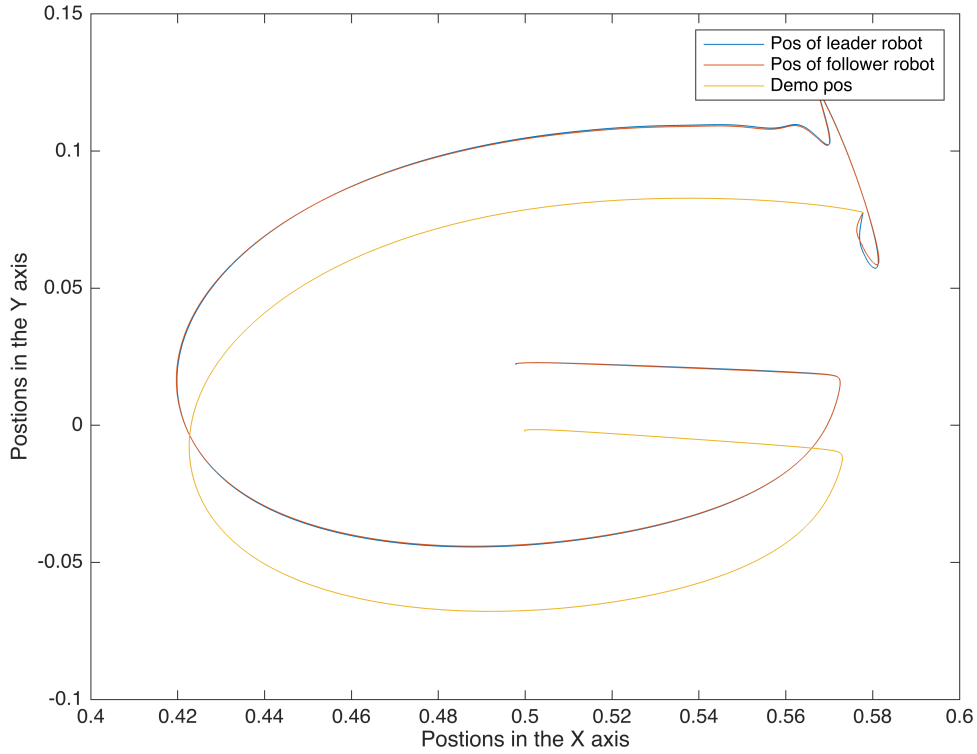
- Save and plot the positions on both the leader and follower sides
- Use these plots in your written analysis; compare the trajectories and discuss disparities between them.
- The control diagram built in Simulink should be included as a screenshot.

```
% This is the example of drawing a image to analyse the results of the PD
control
load('SimOut.mat')
Time = out.tout';
figure,
Joint1 = out.joint.data(:,1);
Joint2 = out.joint.data(:,2);
Joint3 = out.joint.data(:,3);
Joint4 = out.joint.data(:,4);
plot(Time,Joint1); hold on; plot(Time,Joint2); hold on
plot(Time,Joint3); hold on; plot(Time,Joint4);
legend('Joint 1 of leader robot','Joint 2 of leader robot',...
    'Joint 1 of follower robot','Joint 2 of follower robot')
xlabel('Time/s');ylabel('Joint/rad')
```




```
% please compare the results, such as positions and joints for the results
% generate by DMP and those of the leader and follower robots
figure,
start_pos = 1; out.pos.data = out.pos.data;
Pos1_x = out.pos.data(start_pos:end,1); Pos1_y =
out.pos.data(start_pos:end,2);
```

```
Pos2_x = out.pos.data(start_pos:end,3); Pos2_y =
out.pos.data(start_pos:end,4);
Pos3_x = out.pos.data(:,5); Pos3_y = out.pos.data(:,6);
plot(Pos1_x,Pos1_y); hold on;
plot(Pos2_x,Pos2_y); hold on
plot(Pos3_x,Pos3_y);
legend('Pos of leader robot', 'Pos of follower robot','Demo pos')
xlabel('Postions in the X axis')
ylabel('Postions in the Y axis')
```



If you have used MATLAB code to write your controller, add it into the code block below:

If instead you have built your controller using Simulink blocks, add a screenshot of the controller below. If you have used subsystems, you may need multiple screenshots:

Describe and compare your results below (max. 500 words):

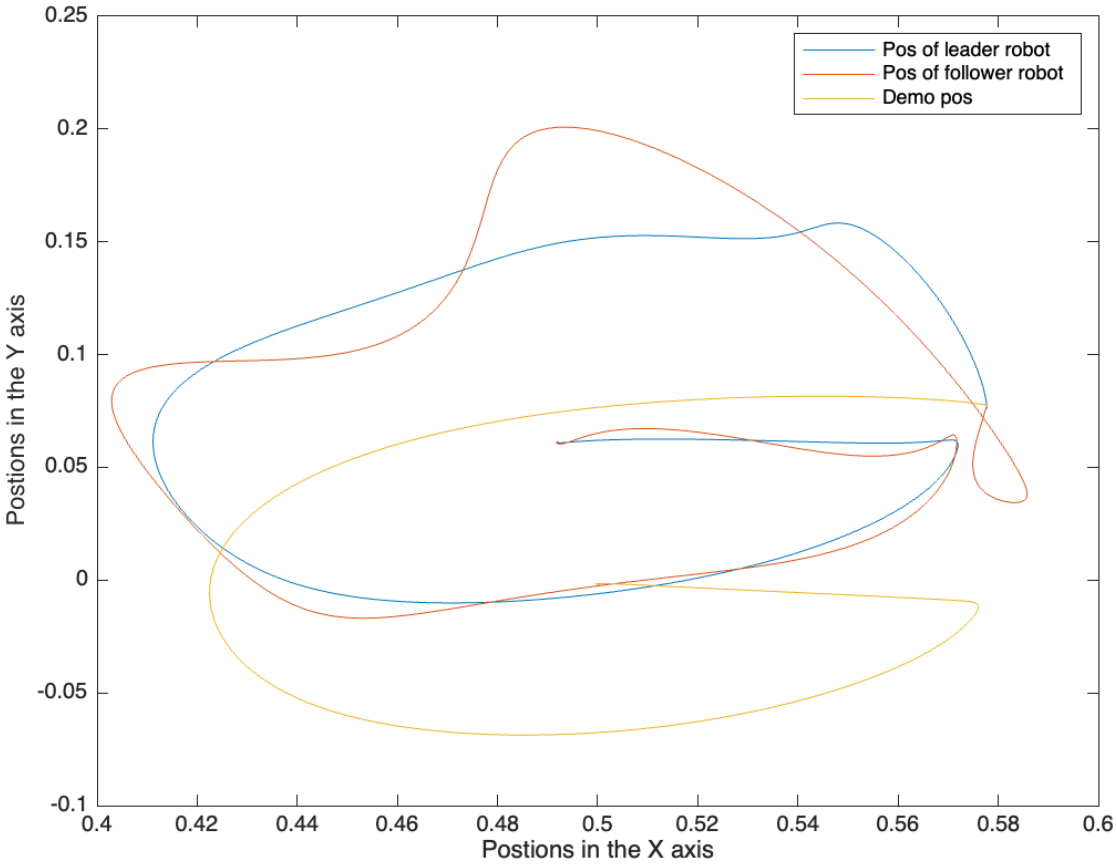
To begin with, we re-calculate the starting points of the leader and follower to match the starting point of the demo setpoint. By extracting the first point (x,y) from improved DMPs' trajectory, we obtained the coordinates (0.077774069268124, 0.077610277810050) mm. Then, we use the *inverse1.m* file to calculate the initial joint

angles of the manipulators, resulting in (6.1778, 0.4777) radians. Then, we modify the x0 values for the leader and follower in the files *master1.m (line 25)* and *slave1.m (line 24)*, respectively. the *starting_pos* variable has change from 1500 to 1, without applying any scaling.

This is the result of the modification.

We can see that the starting points from demo setpoint, leader, and follower are now coincide at the same location.

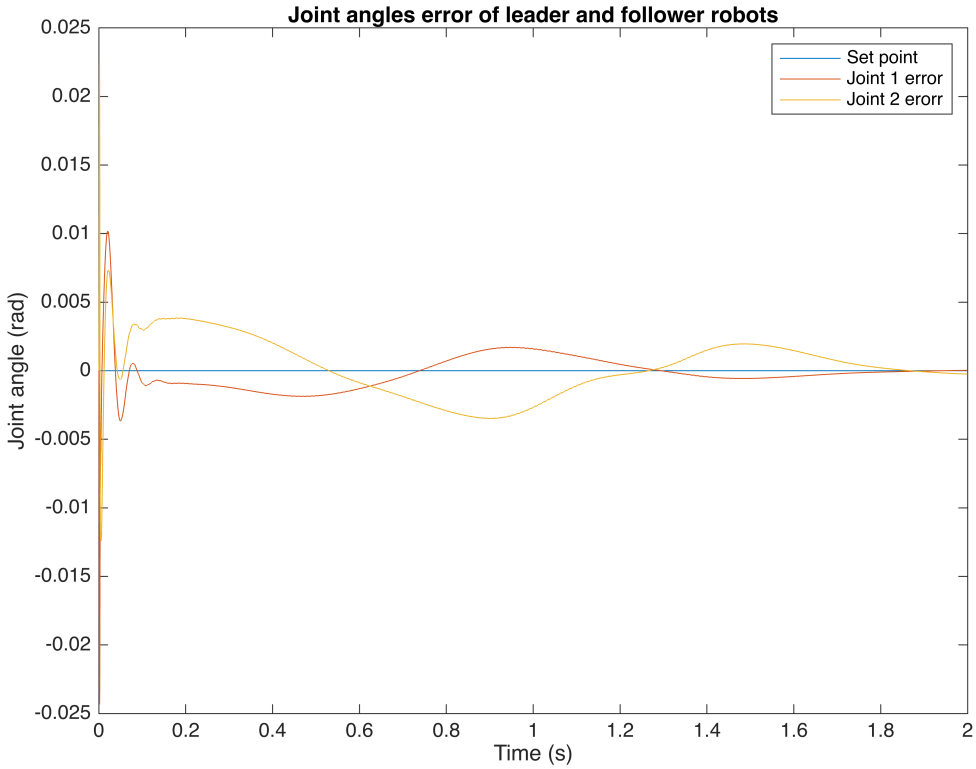
In the file *mas_sc1.m (line 48)* file, *a* PD controller for a model-based portion of the leader is implemented. In *mas_c3.m (line 50)* file, a PD controller for a model-based portion of follower is implemented. This section discusses how to determine the appropriate P and D gains for both controllers by conducting experimenst, ploting the resulting errors, and adjusting the gains accordingly.

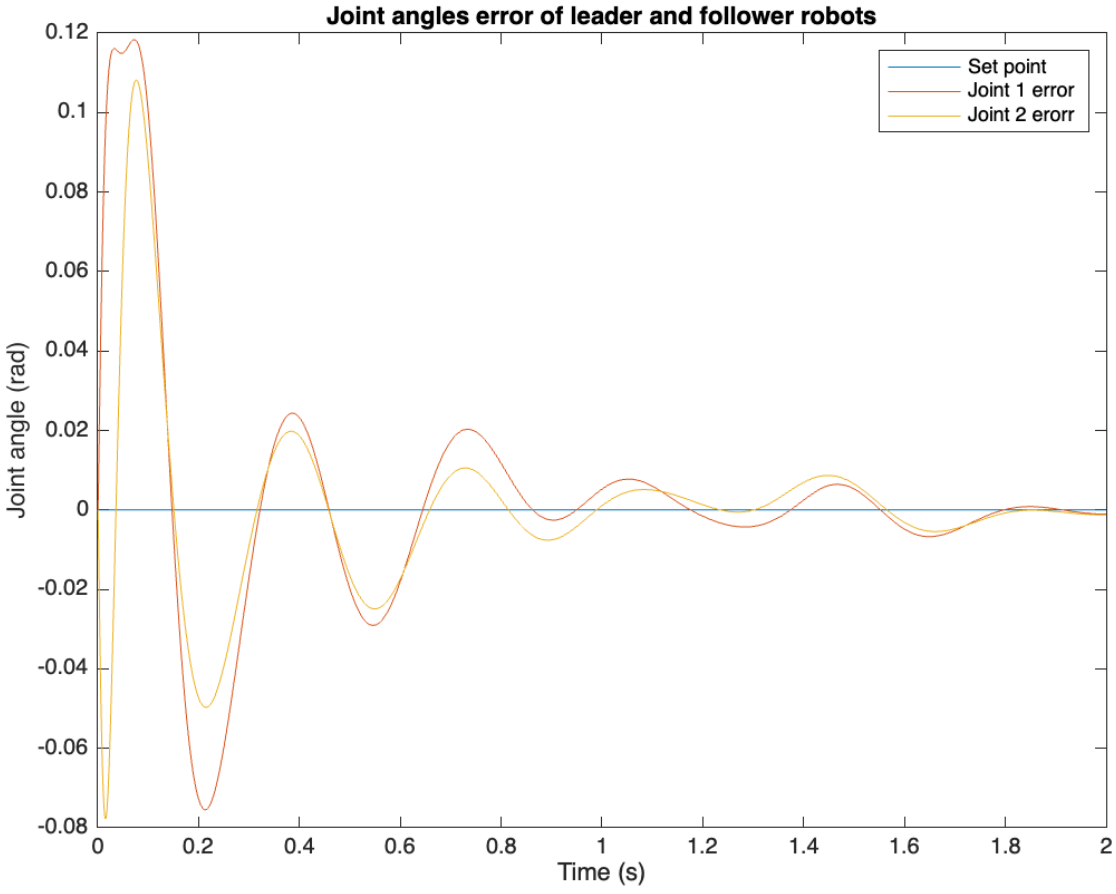


The figure above illustrates that both the leader and follower start from the same configuration. Next, we will tune the PD gains for the follower controller. Let's consider the code snippets below. This code shows the joint error of leader and follower.

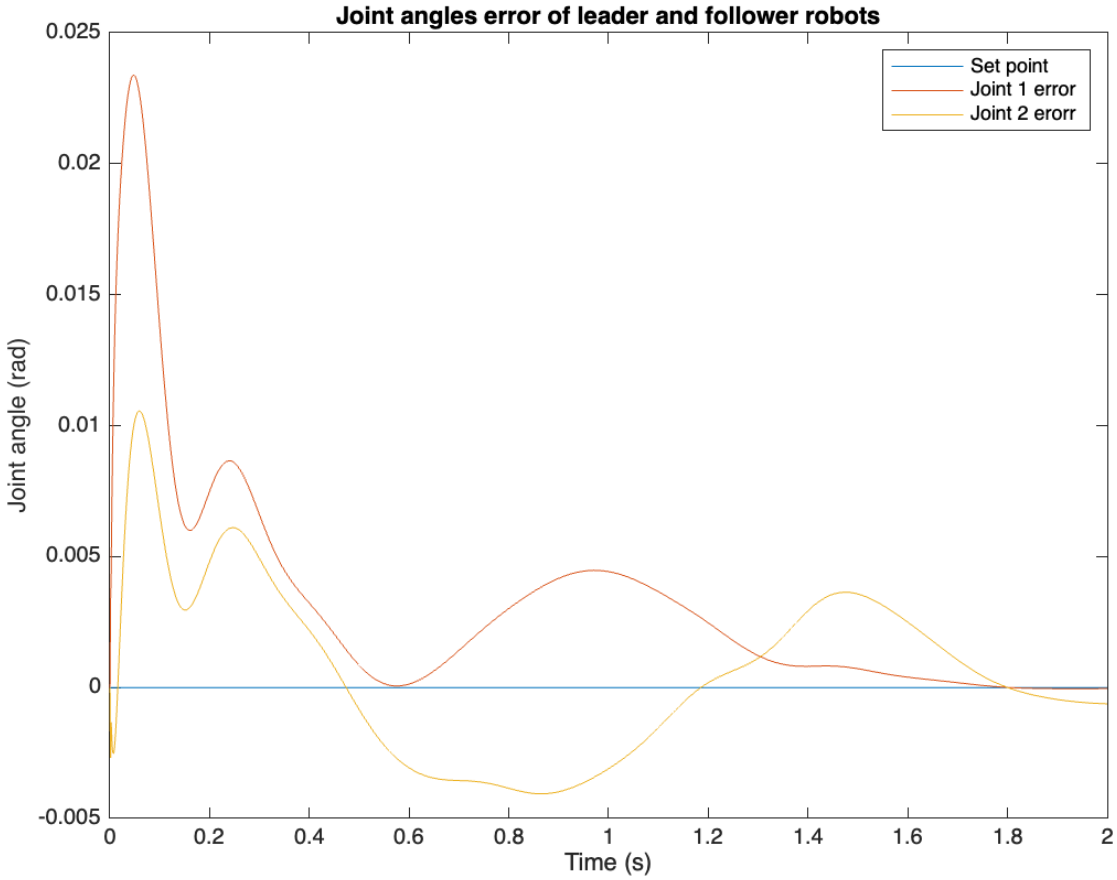
```
% Plot joint error
figure,
Joint1_error = out.joint_error.data(:,1);
Joint2_error = out.joint_error.data(:,2);
plot(Time, zeros(1, length(Joint1_error))); hold on;
plot(Time,Joint1_error); hold on; plot(Time,Joint2_error);
title("Joint angles error of leader and follower robots")
legend('Set point', 'Joint 1 error','Joint 2 erorr')
```

```
xlabel('Time (s)');ylabel('Joint angle (rad)')
```

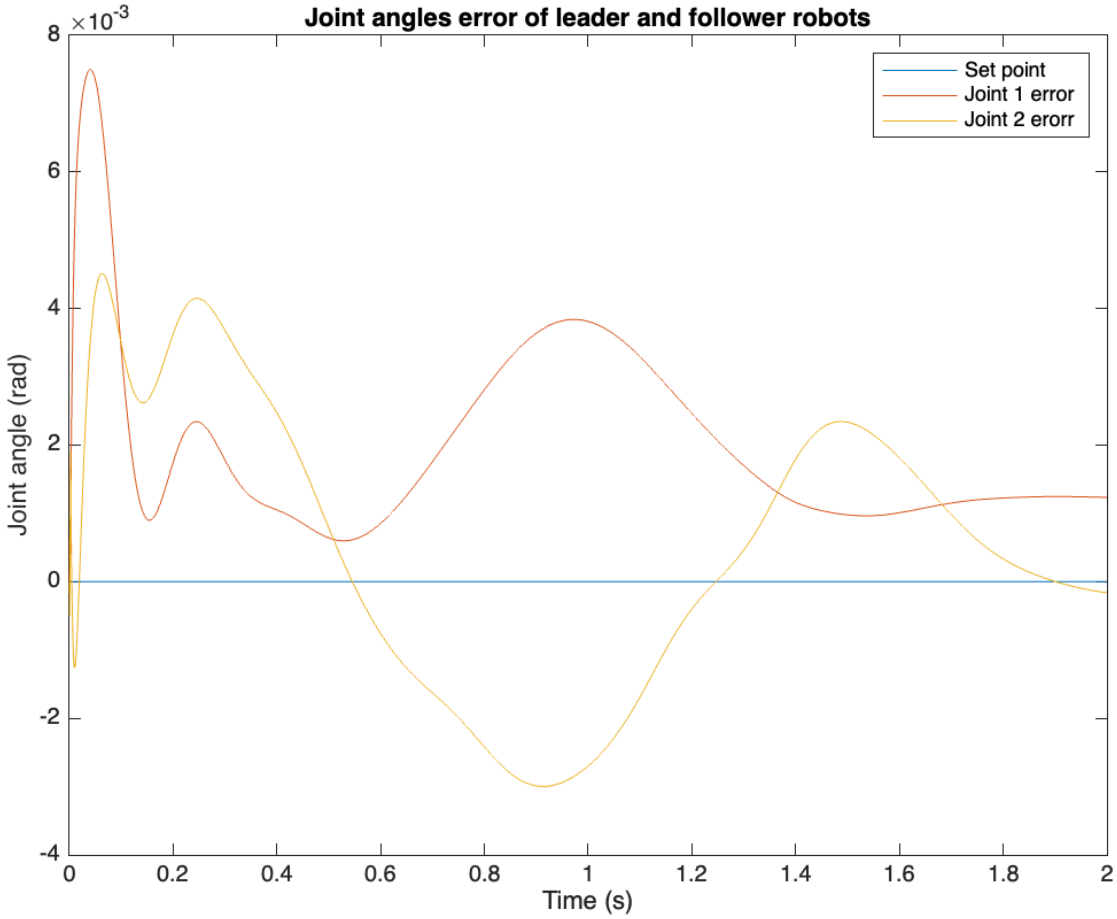




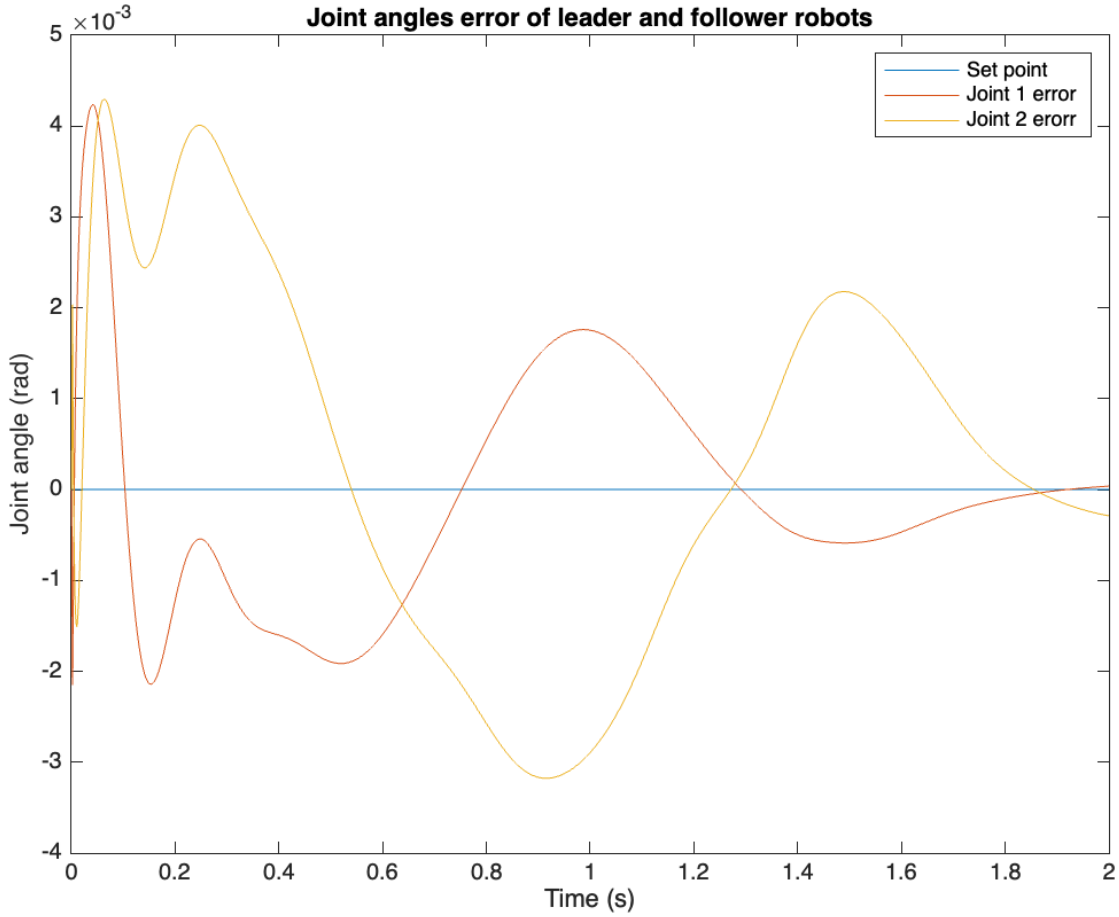
The figure above shows the resulting error when setting the PD controller gains for the leader and follower to **P_gain = 100** and **D_gain = 2**. The plot reveals that the error is oscillatory and takes time to settle. This indicates that the **D_gain** is too low, so we increase it significantly to **50** and observe the effect.

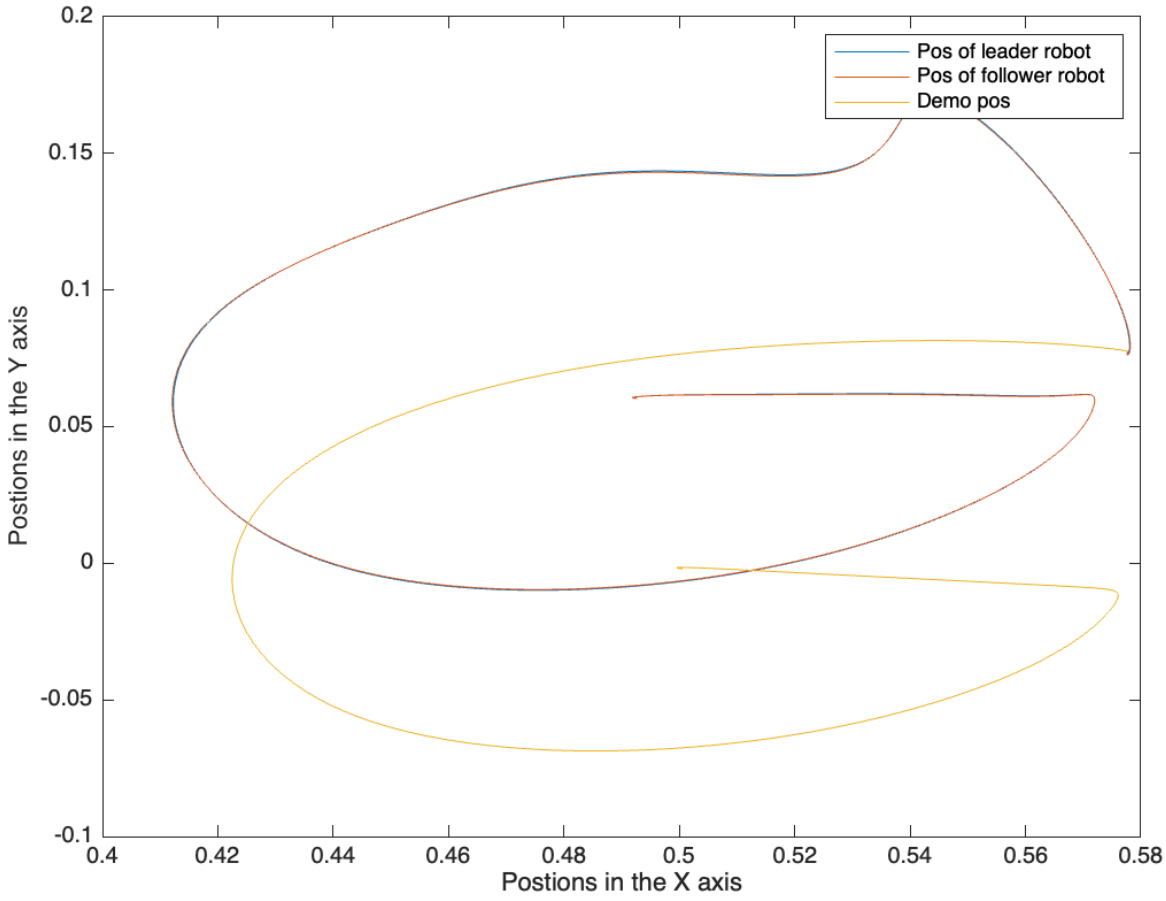


With **D_gain = 50**, the error response appears to settle better. We then increase **D_gain** further to **200**



The response improves noticeably. The error is significantly reduced from **0.025 mm** to **0.008 mm**. However, there is still some overshoot at the beginning, suggesting that the **P_gain** is too high. Therefore, we reduce **P_gain** to **1**.

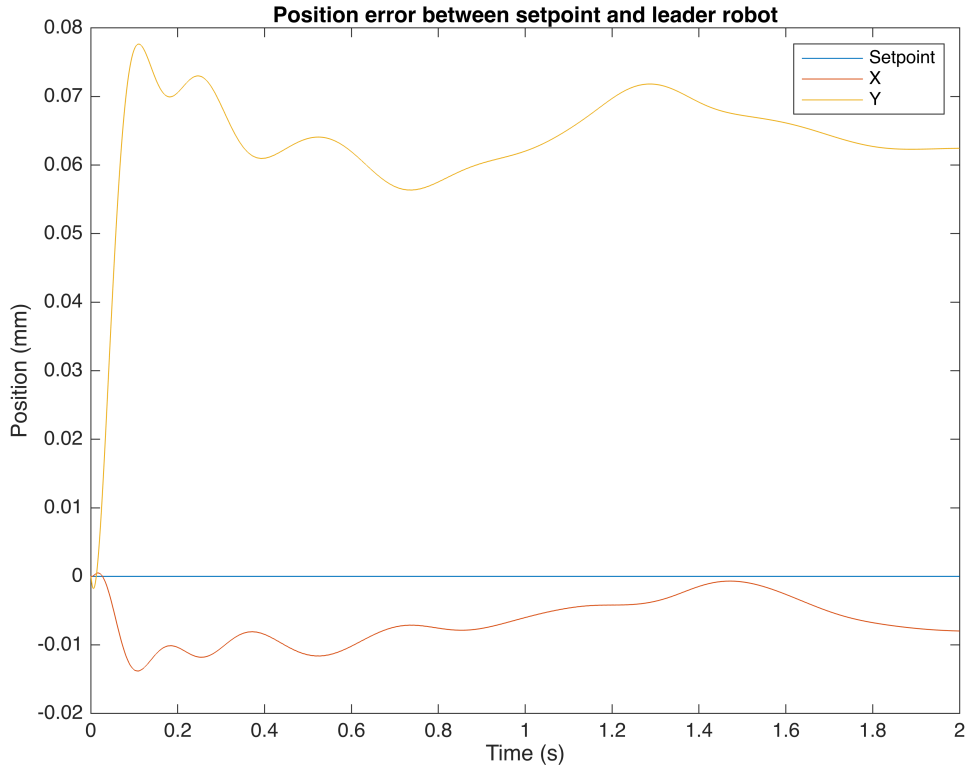

Joint angles error of leader and follower robots
×10-3
Set point
Joint 1 error
Joint 2 erorr
Joint angle (rad)
Time (s)


Pos of leader robot
Pos of follower robot
Demo pos
Postions in the Y axis
Postions in the X axis

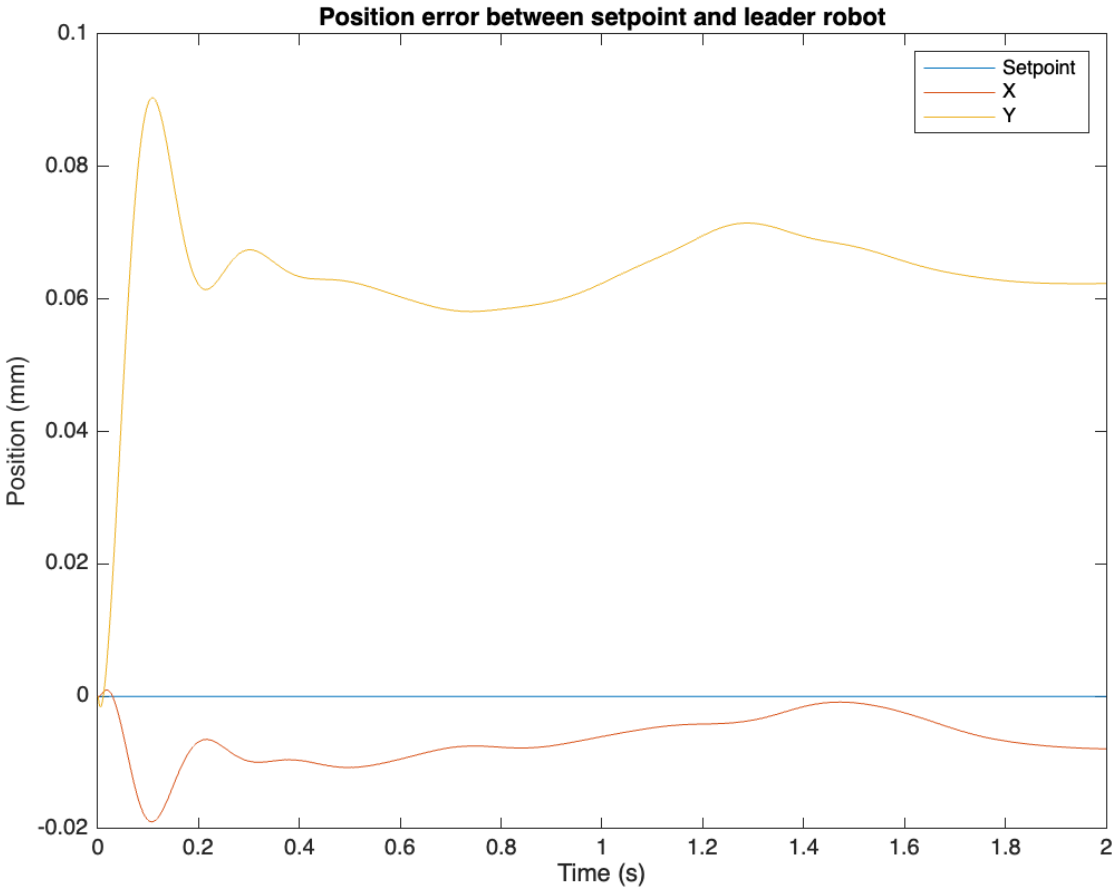
As a result, the overshoot decreases substantially—from **0.008 mm** to **0.005 mm**—and the trajectories of the leader and follower overlap well. While minor oscillations in the error response remain, the trajectory error is now within an acceptable range (**P_gain = 1**, **D_gain = 200**).

Now, we will tune the PD controller of the leader.

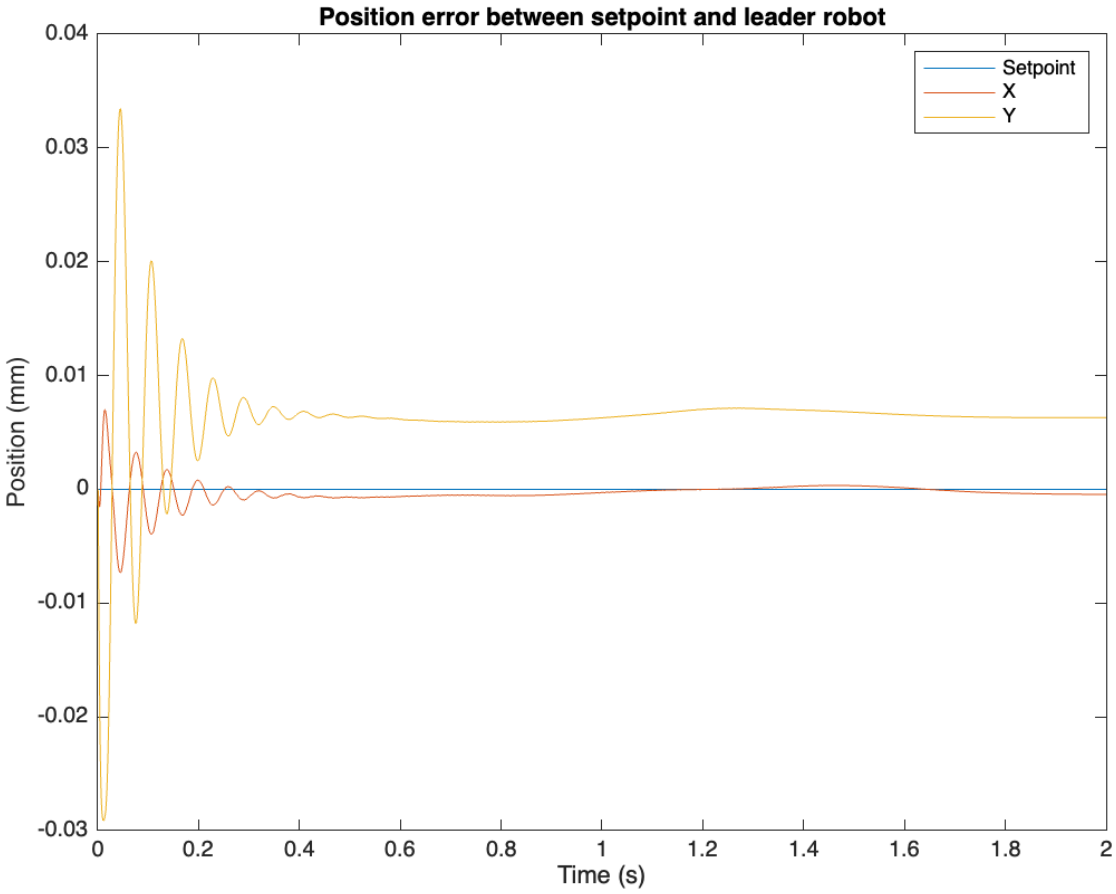
```
% Plot position error
figure,
leader_x_err = out.pos.data(:,1) - out.pos.data(:,5);
leader_y_err = out.pos.data(:,2) - out.pos.data(:,6);
plot(Time, zeros(1, length(leader_x_err))); hold on;
plot(Time,leader_x_err); hold on; plot(Time,leader_y_err);
title("Position error between setpoint and leader robot")
legend('Setpoint','X','Y')
xlabel('Time (s)');ylabel('Position (mm)')
```



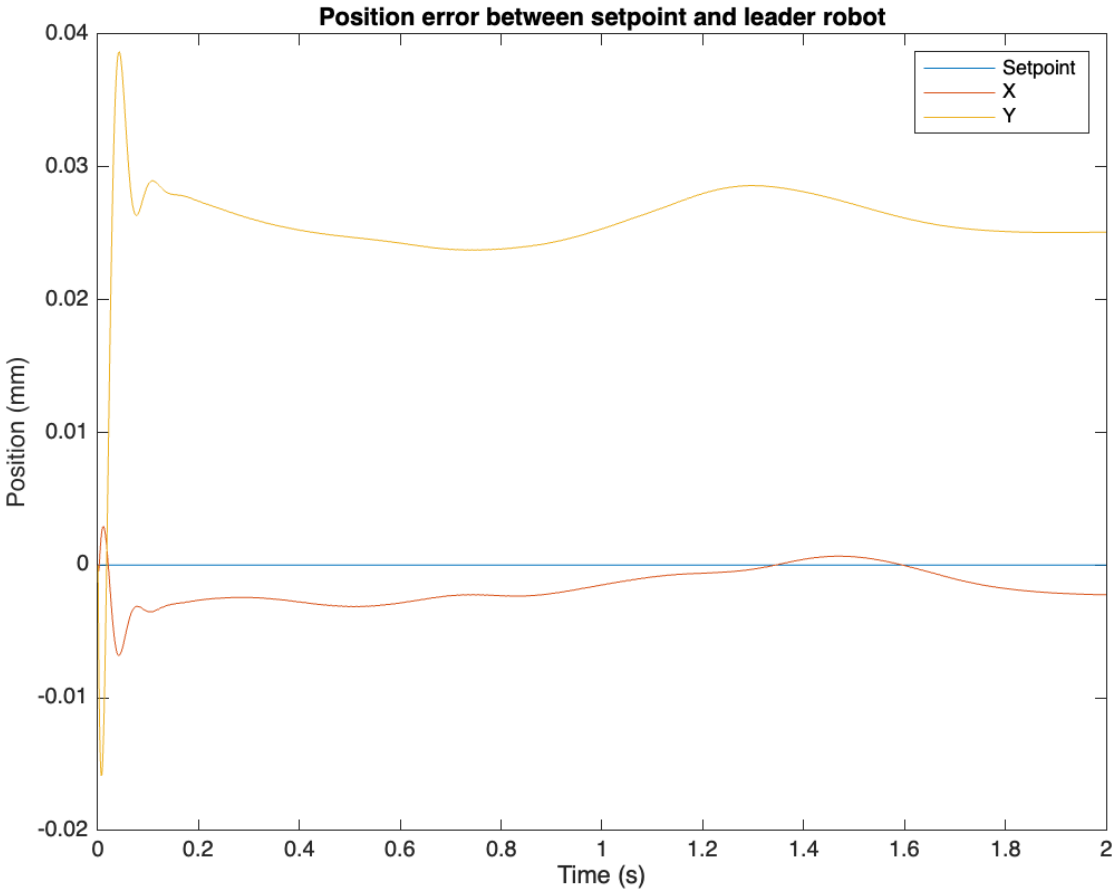
Starting with **P_gain = 100** and **D_gain = 2**, we obtain the error shown in the figure below.



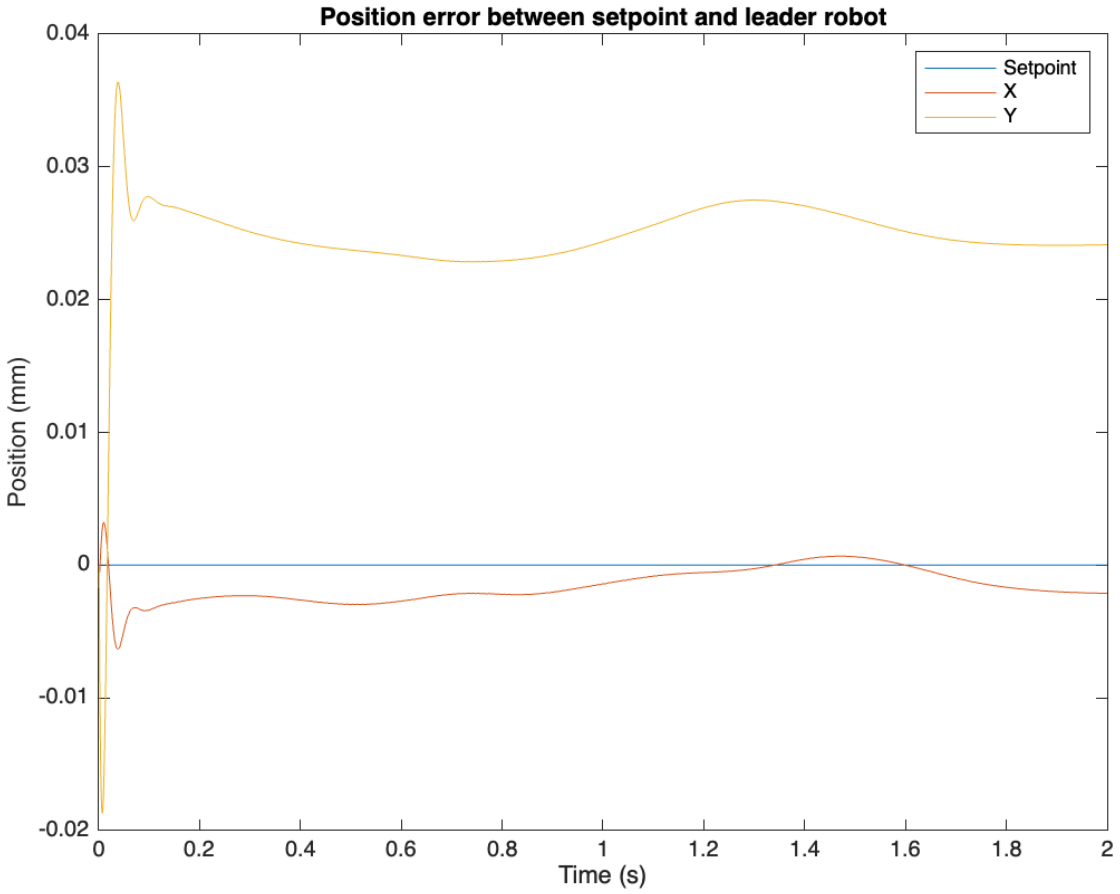
It appears that the steady-state error does not settle. While implementing a **PID controller could address this**, we opt for a simpler approach by increasing the **P_gain**. First, we increase **P_gain** aggressively to **1000** to observe the effect.

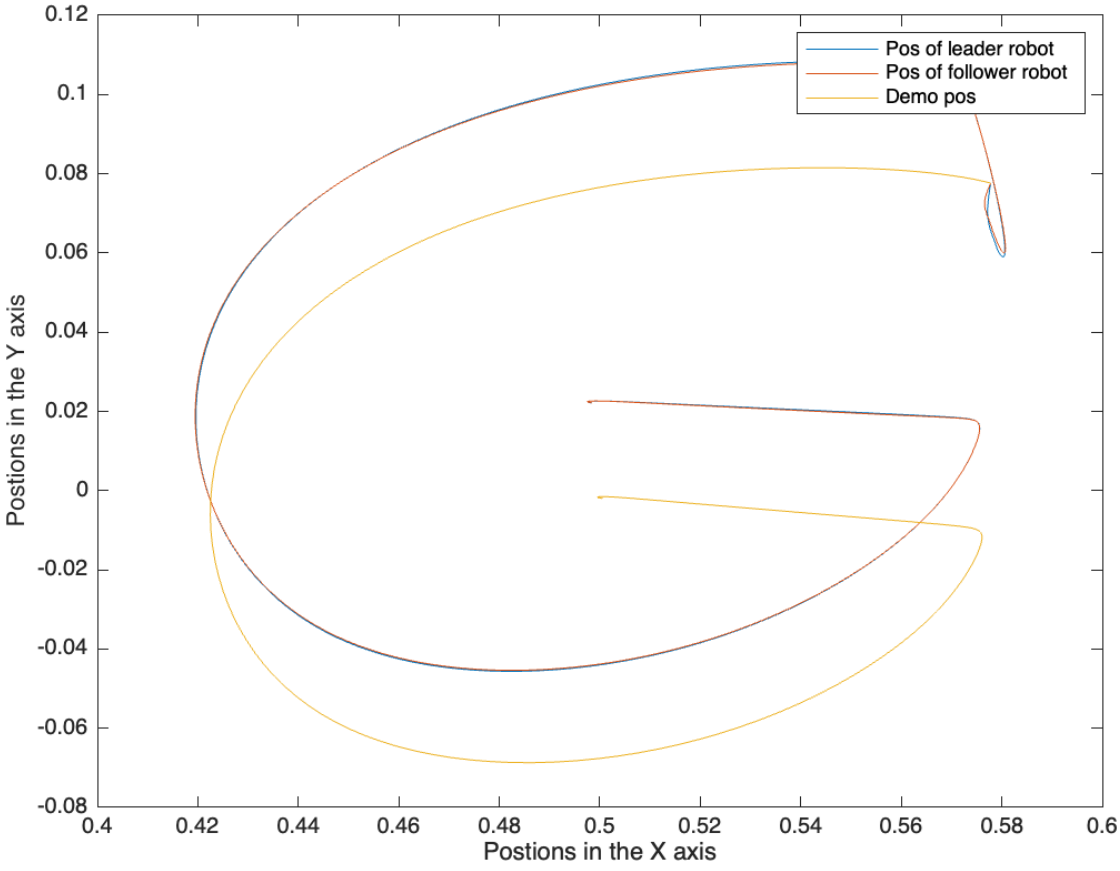


This adjustment seems excessive, but it significantly reduces the steady-state error—from **-0.02 mm** to nearly **0 mm** on the x-axis, and from **0.06 mm** to **0.01 mm** on the y-axis. This is a considerable improvement. Next, we increase the **D_gain** from **2** to **8** to reduce the oscillations.



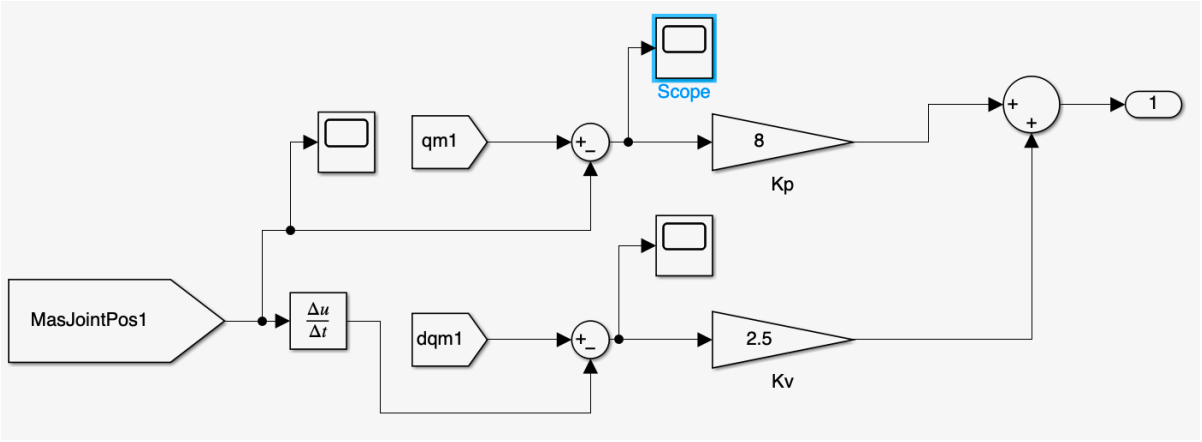
This results in a much better error response. We further increase **P_gain** to **1300** and slightly raise **D_gain** to **10** to see if we can reduce the steady-state error even more.

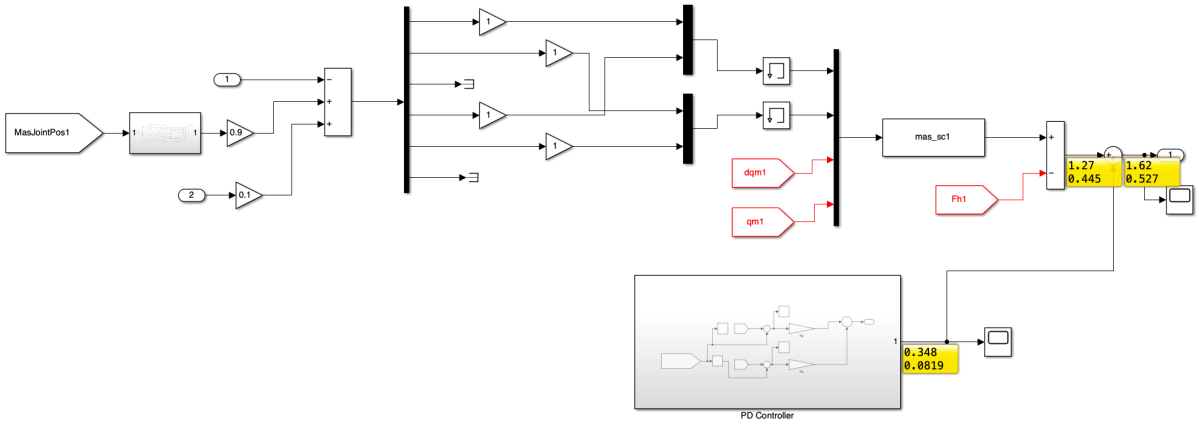

Position error between setpoint and leader robot
Setpoint
X
Y
Position (mm)
Time (s)
0.04
0.03
0.02
0.01
0
-0.01
-0.02
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2


Pos of leader robot
Pos of follower robot
Demo pos
Postions in the Y axis
Postions in the X axis
0.12
0.1
0.08
0.06
0.04
0.02
0
-0.02
-0.04
-0.06
-0.08
0.4
0.42
0.44
0.46
0.48
0.5
0.52
0.54
0.56
0.58
0.6

The error response does not change significantly, but the trajectories appear smoother and more aligned with **P_gain = 1300** and **D_gain = 10**.

Next, we implement PD controller inside the leader model to follow the trajectory as shown in the figures below. The left side is the PD controller implementation by using the joint error between the leader and the generated trajectory. The right side is the integration of PD controller in leader model subsystem.





Extra Credit (max. 10 marks)

Complete the above tasks and document them thoroughly, utilising material covered in the taught material and techniques from the literature. For maximum marks, make sure to:

- Where writing is required, you should describe the your approach to each task justified by the task at hand and your knowledge of the subject - why did you do it this way?
- Make a clear, well-supported comparison of the improved method over the general DMPs; is there a significant improvement (yes or no), and what evidence are you using to support this conclusion?
- Improve your evidence by running multiple variants of your models, such as by changing the parameters in Task 2.
- Improve your analysis by modifying the DMP calculations in Task 3 and detailing the effects of these modification.

- You should also illustrate to what extent the improved method can deal with cooperative robot manipulation.
- Evidence of further reading of the DMP literature, details of the methodology used to approach each task, careful analysis of your results and thoughtful conclusions will all attain additional marks.