IEEE CSS

Date of Submission 12 December 2022

Discrete Neural Network Adaptive Position Control of a 6 Revolute Joint Arm

Connor Jones[1]

[1]Missouri University of Science and Technology, Rolla, MO 65401 USA

CORRESPONDING AUTHOR: Connor Jones (e-mail: cbjd8r@mst.edu)

ABSTRACT One topic that is only briefly discussed in this course, but is of great importance due to the growing accessibility of microcontrollers and therefore rising number of digital control applications, is neural network control of nonlinear systems in a discrete time implementation. This paper will detail the process of discretizing the functional link neural network controller and two-layer neural network controller for a rigid link robot manipulator using methods such as the Euler method and the Taylor Series method. These discretized controllers will then be used to control a 3 revolute joint 2 link elbow arm. The outputs will be compared with the continuous time variant of the controller at multiple different sampling rates to determine if the performance is being significantly degraded. This is being done with the hopes of implementing one of the discrete time controllers on a 6R articulated arm controlled via a microcontroller.

I. INTRODUCTION

Robotic manipulators are key when it comes to interacting with a variety of potential objects. There are many different types of robotic manipulators that all have specific use cases, these are mainly categorized by the types of joints they possess, how many joints they possess, and how those joints are organized. The robotic arm that is going to be used as an example in this paper is an arm that uses all revolute joints, also known as an articulated arm and has a total of six joints. This arm is one of the most universally applicable arms with a broad variety of use cases, which is why it was chosen. Three of the joints will be used for position control and 3 joints will be used for orientation control. A unique aspect of the arm being used is that it possesses a spherical wrist meaning the 3 orientation joints exist in the same point in space, which allows the decoupling of the position and orientation problems. The following paper will describe the process of selecting a neural network adaptive controller and a discretization method in order to control the position of the arm described above using a microcontroller.

It was chosen to only control position because of the use case the controllers and arm are being designed for. The robotic arm will be controlled by a human driver using a remote control transmitted to the robot via a signal. This system is used by a variety of important applications to control manipulators such as on: mars rovers, bomb disposal robots, medical robots, space shuttle maintenance arms, and more. Position controllers are being implemented on this robot to make it much easier to drive as controlling each of the six joints independently is an arduous task and is very slow; however, it was determined that by making position control easier by simply giving a point in space for the manipulator to navigate to, it is not much of a challenge for the driver to manually control the final three orientation joints. In fact, it was easier for the driver to conceptualize moving the orientation angles incrementally by applying torques via the control input than by defining the angles and having a controller move to those angles.

A microcontroller was chosen to run the controller for a variety of reasons. First, most of the applications listed above are mobile robot applications where the weight of the robot is a concern. Microcontrollers are very light and small which is very beneficial in this situation over larger, heavier solutions. Second, microcontrollers are most likely already being used to serve another purpose on the arm such as reading sensor values or parsing the control input signal. Finally, microcontrollers have become much more accessible and much more powerful in roughly the last decade. This gives them the ability to be used in higher performance applications, such as machine learning and adaptive control.

II. BACKGROUND

This section goes over the information needed to understand the three main components of this paper, why we need neural network adaptive controllers and which ones are being tested, discretization methods and why it is important, and the dynamics of a 3R arm which is needed to simulate the controllers.

Table 4.2.1: FLNN Controller for Ideal Case, or for Nonideal Case with PE

Control Input:

$$\tau = \hat{W}^T\phi(x) + K_v r, \ \ with\ \phi(x)\ a\ basis$$

NN Weight/Threshold Tuning Algorithms:

$$\dot{\hat{W}} = F\phi(x)r^T, \ \ with\ F\ a\ positive\ definite\ design\ matrix$$

Table 4.3.1: Two-Layer NN Controller for Ideal Case

Control Input:

$$\tau = \hat{W}^T\sigma(\hat{V}^T x) + K_v r - v,$$

NN Weight/Threshold Tuning Algorithms:

$$\begin{aligned}\dot{\hat{W}} &= F\hat{\sigma}r^T,\\ \dot{\hat{V}} &= Gx(\hat{\sigma}'^T\hat{W}r)^T,\end{aligned}$$

Design parameters: F, G *positive definite matrices.*

Table 4.2.2: FLNN Controller with Augmented Tuning to Avoid PE

Control Input:

$$\tau = \hat{W}^T\phi(x) + K_v r, \ \ with\ \phi(x)\ a\ basis$$

NN Weight/Threshold Tuning Algorithms:

$$\dot{\hat{W}} = F\phi(x)r^T - \kappa F\|r\|\hat{W}$$

Design parameters: F *a positive definite matrix and* $\kappa > 0$ *a small parameter.*

Table 4.3.2: Two-Layer NN Controller with Augmented Backprop Tuning

Control Input:

$$\tau = \hat{W}^T\sigma(\hat{V}^T x) + K_v r - v,$$

Robustifying Signal:

$$v(t) = -K_z(\|\hat{Z}\|_F + Z_B)r$$

NN Weight/Threshold Tuning Algorithms:

$$\begin{aligned}\dot{\hat{W}} &= F\hat{\sigma}r^T - F\hat{\sigma}'\hat{V}^T x r^T - \kappa F\|r\|\hat{W}\\ \dot{\hat{V}} &= Gx(\hat{\sigma}'^T\hat{W}r)^T - \kappa G\|r\|\hat{V}\end{aligned}$$

Design parameters: F, G *positive definite matrices,* $\kappa > 0$ *a small design parameter.*

A. NEURAL NETWORK ADAPTIVE CONTROLLERS

The most commonly used controllers are PID (Proportional Integral Differential) controllers and for many use cases they work within the error bounds necessary; however, their performance degrades when they are used on a system that is non-linear. This was the case when the Mars Rover Design Team attempted to use PID controllers to control their 6R arm, the PID controllers were not giving the performance that was needed to properly control their arm. This was the impetus that drove the search for a better controller, one that was able to maintain a good performance on a nonlinear system. It was also necessary that the controller be able to account for a change in dynamics, for example when the arm picks up an object. The class of controllers that satisfy these constraints are online neural network adaptive controllers, that are specifically designed for use on a robotic arm.

This type of controller uses a neural network to account for the nonlinear behavior of the system and a PD controller to apply the controlling factors. The largest challenge of this controller is training the weights of the neural network. This is because there is no set goal for the output of the neural network and therefore traditional backpropagation cannot be used. It is also paramount that the controller maintain stability during the use of the system as most systems will have large consequences if the controller is unstable. With these two ideas in mind it is possible to design a weight update law using Lyapunov's stability theorem. The process for doing this is described in further detail in [1] along with further information about the design of the controllers. This can be done for both a one layer and a two layer neural network. The design of those controllers is given in the tables 4.3.1 and 4.2.1, both obtained from [1].

The earlier neural net controllers were designed for an ideal system; however, on a non-ideal system they suffer from the persistent excitation condition. This means that the weights are being overtrained and therefore the performance suffers. The controllers can be found in tables 4.3.1 and 4.2.1 and more information on this can again be found in [1].

B. DISCRETIZATION

While there are many benefits described above that microcontrollers have, there is one large drawback: microcontrollers are digital computers, which means they are inherently discrete. The neural net controllers have weight update laws that are in the form of continuous time differential equations. This means that to utilize them on a microcontroller, they must be discretized. This means that they must be made into discrete equations that estimate the continuous time version of the differential equations.

The first step to discretizing a continuous time differential equation is to determine that there is a possible solution. This is easy in this case because one of the steps of the proofs used to determine the weight update laws for all controllers ensures that they are uniformly continuous, and therefore at least locally Lipschitz meaning it is well-posed. There are a few different discretization methods that can be used to estimate a continuous time differential equation. They fall into two main categories, single step and multistep methods. In this case we only need a single step method as we are not estimating the entire differential equation. Using a single step method the next step of the function will be estimated giving us the neural network weights to use during this period, using this updated neural network the controller will apply a torque to the joints. This will bring us to the next cycle where the new state of the system will be used to give a new weight update equation and the process will repeat.

There are four main methods of discretization: the Euler method, the Taylor method, the Backward Euler and higher order backward difference methods, and Higher order Runge-Kutta methods. In this instance the only methods that are of interest are the Euler method and the Taylor method. This is because both the Backward Euler and higher order backward difference methods, and Higher order Runge-Kutta methods use the differential equation at a time that is in

the future from the current time. Because the weight update laws use feedback from the system, that information is not available. Therefore those methods cannot be implemented. Information about discretization in this section was obtained from [2].

1) EULER METHOD

The Euler method is derived from the definition of a derivative, which can be defined at a point a in (1). It is a simple way to estimate the next step in a differential equation but because of that it is not always the most accurate.

$$y'(a) = \lim_{h \to 0} \frac{y(a+h) - y(a)}{h} \tag{1}$$

This can be estimated by replacing h with the time step within the discrete environment and the point a with time t_0. This gives us the equation (2).

$$y'(t_i) \approx \frac{y(t_{i+1}) - y(t_i)}{\Delta t} \tag{2}$$

Then by replacing $y(t_i)$ with it's estimation Y_i and defining $y'(t_i)$ as our differential equation $f(t_i, Y_i)$ it is possible to find an equation for the approximation in (2) which is defined in (3).

$$f(t_i, Y_i) \ = \frac{Y_{i+1} - Y_i}{\Delta t} \tag{3}$$

This can then be solved for Y_{i+1} to obtain the Euler method of discretization in (4).

$$Y_{i+1} = Y_i + \Delta t f(t_i, Y_i) \tag{4}$$

2) TAYLOR METHOD

The Taylor Method is similar to the Euler method, but it is more accurate since it accounts for higher order terms that the Euler method ignores in its estimation. The Taylor method is found using the Taylor series expansion of $y(t_i + \Delta t)$ which is shown in (5).

$$y(t_i + \Delta t) = y(t_i) + \Delta t y'(t_i) + \frac{(\Delta t)^2}{2!} y''(t_i) + \ldots + \frac{(\Delta t)^k}{k!} y^{[k]}(t_i) \tag{5}$$

It is possible to find the Taylor method estimation for any order of estimation but for this application it will just be estimated to the second order because it becomes too resource intensive to calculate between the desired time step and provides less error correction the higher the order. It is also interesting to point out that the Euler method can also be found by finding the first order Taylor method.

To find the second order estimation the terms higher than second order are dropped. $y'(t_i)$ can then be replaced by the differential equation and $y''(t_i)$ with (6) which is found using the chain rule.

$$y''(t) = \frac{\partial f}{\partial t}\frac{dt}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt} = f_t + f_y f \tag{6}$$

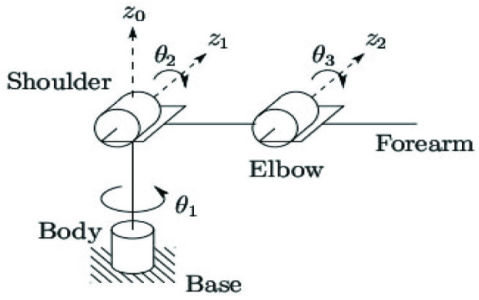

FIGURE 1. Example 3R arm

This makes it possible to obtain (7).

$$y(t_i + \Delta t) \approx y(t_i) + \Delta t f(t_i, y(t_i)) + \frac{(\Delta t)^2}{2!}[f_t(t_i, y(t_i)) + f(t_i, y(t_i)) f_y(t_i, y(t_i))] \tag{7}$$

Like in the Euler method by replacing $y(t_i)$ with its estimate we can obtain an equation for the approximation of the next step and the final version of the second order Taylor approximation (8).

$$Y_{i+1} = Y_i + \Delta t f(t_i, Y_i) + \frac{(\Delta t)^2}{2!}[f_t(t_i, Y_i) + f(t_i, Y_i) f_y(t_i, Y_i)] \tag{8}$$

C. 3R ARM DYNAMICS

An important step to simulating the performance of the controllers is the ability to have an accurate model to test on, this means that the dynamics of the arm must be found in order to properly simulate the system. The dynamics will only be found for a 3R arm, because, as discussed above, the controllers are only responsible for controlling the position of the arm and since there is a spherical wrist the last three joints can be removed without changing the position problem. An example of this arm can be seen in fig. 1. The general form of a robotic manipulator with rigid links is written as (9).

$$M(q)\ddot{q} + V_m(q, \dot{q})\dot{q} + F(\dot{q}) + G(q) + \tau_d = \tau \tag{9}$$

Where M is the mass matrix, V_m is the Coriolis/centripetal matrix, F is the friction matrix, τ_d is the applied disturbance torques vector, and q is the joint angle vector. The values of these can be found for a specific arm by using Lagrange equations (10), where K and P indicates the kinetic and potential energy respectively.

$$\begin{aligned} L &= K - P \\ \frac{d}{dt}\frac{\partial L}{\partial \dot{q}} - \frac{\partial L}{\partial q} &= \tau \end{aligned} \tag{10}$$

To find the kinetic and potential energy it is necessary to first convert the joint angles into Cartesian positions. This will be first done for the first link, these can be seen in (11).

$$\begin{aligned} x_1 &= cos(q_1)(a_1 cos(q_2)) \\ y_1 &= sin(q_1)(a_1 cos(q_2)) \\ z_1 &= a_1 sin(q_2) \end{aligned} \tag{11}$$

From these positions it is not possible to find all the link velocities by taking the derivative of the positions in (11). The result is shown in (12).

$$\begin{aligned} \dot{x_1} &= -a_1(\dot{q_1} sin(q_1)cos(q_2) + \dot{q_2} cos(q_1)sin(q_2)) \\ \dot{y_1} &= a_1(\dot{q_1} cos(q_1)cos(q_2) - \dot{q_2} sin(q_1)sin(q_2)) \\ \dot{z_1} &= a_1 \dot{q_2} cos(q_2) \end{aligned} \tag{12}$$

The velocities and position will then be used to find the kinetic and potential energy of the first link, the results of this are in (13) and (14).

$$K_1 = \frac{1}{2} m_1 v_1^2 = \frac{1}{2} m_1 (\dot{x_1}^2 + \dot{y_1}^2 + \dot{z_1}^2) \tag{13}$$

$$P_1 = m_1 g z_1 = m_1 g a_1 sin(q_2) \tag{14}$$

The process can then be repeated to find the kinetic and potential energy for the second link as well since we need to find the total kinetic and potential energy of the system. The process for that can be viewed below.

$$\begin{aligned} x_2 &= cos(q_1)(a_1 cos(q_2) + a_2 cos(q_2 + q_3)) \\ y_2 &= sin(q_1)(a_1 cos(q_2) + a_2 cos(q_2 + q_3)) \\ z_2 &= a_1 sin(q_2) + a_2 sin(q_2 + q_3) \end{aligned} \tag{15}$$

$$\begin{aligned} \dot{x_2} &= -\dot{q_1} sin(q_1)(a_1 cos(q_2) + a_2 cos(q_2 + q_3)) \\ &- cos(q_1)(a_1 \dot{q_2} sin(q_2) + a_2(\dot{q_2} + \dot{q_3}) sin(q_2 + q_3)) \end{aligned} \tag{16}$$

$$\begin{aligned} \dot{y_2} &= \dot{q_1} cos(q_1)(a_1 cos(q_2) + a_2 cos(q_2 + q_3)) \\ &- sin(q_1)(a_1 \dot{q_2} sin(q_2) + a_2(\dot{q_2} + \dot{q_3}) sin(q_2 + q_3)) \end{aligned} \tag{17}$$

$$\dot{z_2} = a_1 \dot{q_2} cos(q_2) + a_2(\dot{q_2} + \dot{q_3}) cos(q_2 + q_3) \tag{18}$$

$$K_2 = \frac{1}{2} m_1 v_2^2 = \frac{1}{2} m_2 (\dot{x_2}^2 + \dot{y_2}^2 + \dot{z_2}^2) \tag{19}$$

$$P_2 = m_2 g z_2 = m_2 g[a_1 sin(q_2) + a_2 sin(q_2 + q_3)] \tag{20}$$

Once the total Lagrangian value is found you can utilize Lagrange's equation, shown in (10), to find the dynamics of the system. The result of this process is shown in the code in the appendix and in [3], but it is not shown here as it is too long to justify including.

III. METHODOLOGY

The goal of the simulation process is to determine the best combination of controller and discretization method to employ on the microcontroller controlling the position of a six degree of freedom articulated robotic arm. It was because of this that it was chosen to test the performance on a set of desired joint angles to make the arm perform from a real possible use case - tracing a square in a 2D plane in front of the arm. This could emulate interacting with devices on a service panel of some sort. The desired joint angles were obtained using a program written by a previous member of the Mars Rover Design Team and can be found in [4]. The inverse kinematic program produces a list of joint angles through time that when applied to the robotic arm make the arm follow a desired path, the way it does this is beyond the scope of this paper. This output is a discrete signal; however, the time step can be varied and a very small time step on the order of 10^-5 seconds was chosen so that the signal was essentially continuous.

The four continuous time controllers described above were then implemented in MATLAB, along with a classical PD controller. These five controllers were then simulated using the desired joint angles and the dynamics of the 3R arm described above using the ODE45 solver in MATLAB. The weight update differential equations were then taken out of the ODE45 solver and instead maintained by the discretization methods: Euler method, Partial 2nd Order Taylor Method, and Full 2nd Order Taylor Method. Partial Order Taylor Method means utilizing the Taylor method, but leaving out the time derivative of $f(t, Y_i)$, the equation for this method can be seen in (21).

$$Y_{i+1} = Y_i + \Delta t f(t_i, Y_i) + \frac{(\Delta t)^2}{2!}[f(t_i, Y_i) f_y(t_i, Y_i)] \tag{21}$$

This was tested because it is much easier to compute as many of the matrices already have to be computed in the continuous time variation, therefore very calculations need to be done but some of the benefits of the Taylor method are still gained. The time derivative is much more complicated to produce and requires the use of the chain rule to calculate the partial time derivative with respect to the multiple changing factors in the weight update law. This may increase the necessary duration of the time step, which could have a bigger negative impact on performance than using a simpler discretization method. It is also prevalent to point out that, because of this same reason, the two layer neural network controllers that are categorized as being discretized using the Full 2nd Order Taylor Method have first layer weight update laws that only use partial Taylor method as the time derivative of the differential equation used to update the first layer weights was too complex to compute within the desired time step.

These controllers were then tested on a variety of different time step sizes to see to what degree a larger time step would negatively affect the performance of the controller. To test the robustness of the controllers a vertical step was added in at 10 seconds, making the signal non-continuous. To further test the robustness of the controller the robot was also simulated to pick up an object at 14 seconds by increasing the weight at the end of the arm by 1.5kg. The performance of the controller was based on the average absolute error of the desired joint angles vs the actual joint angles throughout the length of the trial.

IV. SIMULATION

The 3R arm was simulated with a link 1 weight of 0.8 kg, link 2 weight of 2.3 kg, and a payload weight of 2.7 kg to start with an addition of 1.5 kg at 14 seconds. The lengths of the links are both 1 m. The acceleration of gravity is the standard 9.8 m/s^2. Like mentioned in the methodology this arm was controlled by five different controllers: one layer neural net, one layer neural net with augmented tuning to avoid persistent excitation, two layer neural net, two layer neural net with augmented tuning to avoid persistent excitation, and a classic PD controller. In the results tables 1-4 these are labeled 1LNN, 1LNNAT, 2LNN, 2LNNAT, and PD respectively.

The results tables show the absolute average error during each of the trials for the most interesting time steps. Note that each of the discrete tables have a timestep and a number for the number of torque updates per time step. The timestep indicates how much time is in between each update of the weights, while the torque update per time step value indicates how many times the un-modified neural network is used to generate a new output torque during each of the weight update periods. The idea behind this setup is that it is more resource intensive to update all of the weights of the neural network than to use the neural network to generate a new torque. It is also not entirely necessary to update the neural network as frequently as generating new torque outputs since once the neural network learns the dynamics of the arm, which is accounted for using the neural net weights, they change much less frequently than the state of the arm, which is controlled with joint torques. This allows the microcontroller to lower the amount of computations necessary with the goal of making the timesteps between torque updates as small as possible while still updating the neural network to properly account for the nonlinearities of the arm and any change in dynamics.

None of the discrete controllers quite matched the performance of their continuous time counter parts, which is to be expected as to change the controllers to their discrete versions an approximation needed to be used. The process of doing these approximations is discussed in the background section. This resulted in the introduction of more errors. The PD controller served as the baseline for each of the tests, it did not account for the nonlinearities of the system at all

Table 1

	Euler	Partial Taylor	Full Taylor
$\Delta t = 0.02$		Torque update = 100 /Δt	
2LNN	0.0403	0.0403	0.0410
2LNNAT	0.0131	0.0132	0.0132
1LNN	--	--	--
1LNNAT	0.0442	0.0437	0.0455
PD	0.2329	0.2329	0.2329

Table 2

	Euler	Partial Taylor	Full Taylor
$\Delta t = 0.002$		Torque update = 10 /Δt	
2LNN	0.0503	0.0503	0.0551
2LNNAT	0.0201	0.0201	0.0207
1LNN	--	--	0.2257
1LNNAT	0.0618	0.0612	0.0604
PD	0.2329	0.2329	0.2329

Table 3

	Euler	Partial Taylor	Full Taylor
$\Delta t = 0.001$		Torque update = 10 /Δt	
2LNN	0.0499	0.0499	0.0489
2LNNAT	0.0238	.0237	0.0241
1LNN	0.0785	.0785	0.0911
1LNNAT	0.0737	.0740	0.0701
PD	.2325	.2325	0.2325

Table 4

1LNNAT	2LNNAT	1LNN	1LNNAT	1LNNAT
0.0445	0.0117	0.0479	0.0629	0.1603

Continuous time absolute errors

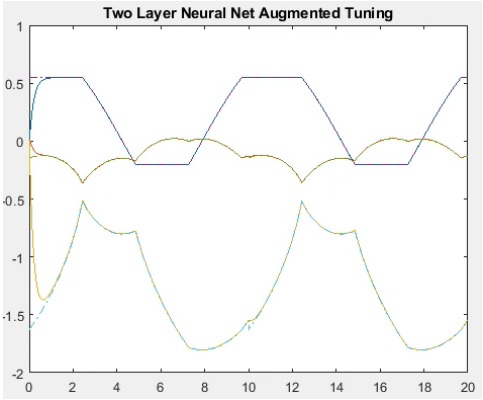

FIGURE 2. **2 Layer Neural Net with augmented tuning with $\Delta t = 0.02$ and a torque update of 100 per Δt**

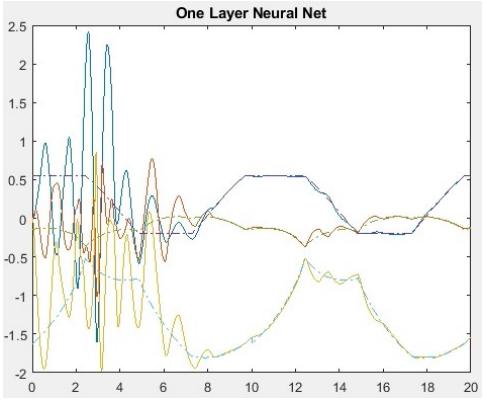

FIGURE 3. **1 Layer Neural Net with $\Delta t = 0.002$ and a torque update of 10 per Δt**

and therefore performed the worst in all test cases, discrete or continuous. Though it is much simpler to implement as it does not train a neural network, if the microcontroller being used is not powerful enough to train a neural network and the error is acceptable this could be a solution. The augmented tuning method consistently outperformed the standard tuning method, there is not a discrete or continuous trial that this is not the case. This is because the system is not ideal therefore persistent excitation needs to be avoided. In addition, in testing the controllers that avoided persistent excitation were much better at handling the step introduced at 10 second into the trial. It also does not require much compute power to add the extra term that avoid persistent excitation, $-\kappa F||r||\hat{W}$ or $-\kappa G||r||\hat{V}$, that it will nearly always be worth the extra compute power for the added performance benefit and stability.

The best performing controllers overall were the two layer neural network class. The added layer compared to the single layer neural network gave it the ability to better account for the dynamics of the system and it allowed for the dynamics to be accounted for more quickly which helped when the time step was increased as there was a more meaningful modification to the weights each time step compared to the the single layer controllers. The main drawback of this type of controller; however, is that it is much more resource intensive, requiring many more weights to be maintained. For example in this simulation the single layer network needed to maintain 33 weights since there were 10 neurons in the hidden layer and 3 outputs, a torque for each joint. The 2 layer controllers had to maintain 193 weights. 160 from the first layer, 16 inputs to each of the 10 hidden layer neurons, plus the 33 from the output layer. In addition to the number of extra weights the weight update laws are more complicated as well for the first layer since the chain rule for derivatives must be used to find the contribution to the error from the first layer weights. If the microcontroller used is powerful enough to handle all of this the two layer neural networks should be used. An example of the 2 layer neural network controller can be seen in figure 2. This case also had the most time in between weight updates which makes implementing the more resource intensive controller more feasible.

The discretization method overall did not make much of an impact overall. In most cases the higher order terms that the Taylor method accounts for does not impact the controller enough to make a significant difference. The only cases that it did make a large difference was when the controller was oscillating significantly. This occurred when the single layer controller, which is already prone to some oscillation, was used with a larger time step and was therefore approximated poorly. The full Taylor method was able to control the transient oscillations and ultimately converge to the desired joint angles, this is shown in figure 3. The other discretization methods were not able to.This is because when the system is accelerating and decelerating this rapidly the higher order terms that are accounted for in the Taylor method play a larger role. However, in the actual unsimulated system it should never see this kind of movement so the benefits of the Taylor method are unlikely to be seen and it is more resource intensive to compute.

V. CONCLUSION

In the simulation section it was shown that the 2 layer neural network with augmented tuning was the best performing controller and that the Euler method performed similarly to the Taylor method in most cases and is easier to compute. Luckily the microcontrollers that the Mars Rover Design Team has access to are some of the fastest on the market with a 600 MHz clock speed and should be able to handle running a 2 layer neural network at a time step within the bounds simulated above. If it is not, the single layer neural network controller with augmented tuning was the second best choice outside the two layer class and that will definitely be able to be handled by the microcontroller. The next step in fully controlling the arm is to implement the MATLAB versions of the controller in C so that it will be able to run on the microcontroller efficiently and then eventually test the controllers on the actual system.

REFERENCES

[1] F. L. Lewis, S. Jagannathan, and A. Yesildirek, *Neural Network Control of Robotic Manipulators and Nonlinear Systems*. Philadelphia, PA, USA: Taylor and Francis, 1999, pp. 175-218. [Online]. Available: https://lewisgroup.uta.edu/FL%20books/Lewis_Jagannathan_Yesildirek%20-%20neural%20network%20control%201999.pdf.

[2] J. Peterson, *Introduction to Discretization*. pp. 1-44. [Online]. Available: https://people.sc.fsu.edu/ jpeterson/IVP.pdf

[3] S. Pezeshki and S. Badalkhani and A. Javadi, "Performance Analysis of a Neuro-PID Controller Applied to a Robot Manipulator," *International Journal of Advanced Robotic Systems*, vol. 9, no. 5, pp. 163, 2012. DOI. 10.5772/51280.

[4] S&T Mars Rover Design Team, *IKModel Matlab Development*. [Online]. Available: https://github.com/MissouriMRDT/IKModel_MatLab_Development

ACKNOWLEDGMENT

I would like to thank Dr. Jagannathan Sarangapani for helping me through this course and providing the knowledge necessary to complete this project.

APPENDIX

Discrete Main Program

```
clear all;
clc;
global angles
global angles_t
global I
global L
global Outs
global NN_delta_t
global PD_delta_t
global tau;
global PD_step_num
global NN_step_num
NN_delta_t = 0.02;
PD_delta_t = NN_delta_t/80; %0.059/5 is max step size
tspan=[0, 20];
angles=load("joint_angles.mat");
angles=pi*angles.q/180;
angles = [angles ; angles];
angles_len=length(angles);
angles_t=0:((tspan(2)+(tspan(2)/angles_len))/angles_len):tspan(2);
I = 16; % Num of NN inputs
L=10; % Num hidden layer neurons
L=L+1;
Outs=3; % Num NN outputs
x0_PD = [0 0 0 0 0 0]';
tau=0;
PD_step_num=0;
NN_step_num=0;
[t_2LNN,x_2LNN]=ode45('robadapt2LNN',tspan,x0_PD);
tau=0;
PD_step_num=0;
NN_step_num=0;
[t_2LNNAT,x_2LNNAT]=ode45('robadapt2LNNAT',tspan,x0_PD);
tau=0;
PD_step_num=0;
NN_step_num=0;
[t_NN,x_NN]=ode45('robadaptNN',tspan,x0_PD);
tau=0;
PD_step_num=0;
NN_step_num=0;
[t_NNAT,x_NNAT]=ode45('robadaptNNAT',tspan,x0_PD);
tau=0;
PD_step_num=0;
NN_step_num=0;
[t_PD, x_PD]=ode45('robadaptPD', tspan,x0_PD);
figure(1)
plot(t_2LNN,x_2LNN(:,1:3),angles_t,angles(:,1:3),'-.');
title("Two Layer Neural Net");
angles_idx=round(t_2LNN/angles_t(2))+1;
error = angles(angles_idx,1:3) - x_2LNN(:,1:3);
mean_2LNN = mean(abs(error));
figure(2)
plot(t_2LNNAT,x_2LNNAT(:,1:3),angles_t,angles(:,1:3),'-.');
title("Two Layer Neural Net Augmented Tuning")
angles_idx=round(t_2LNNAT/angles_t(2))+1;
error = angles(angles_idx,1:3) - x_2LNNAT(:,1:3);
mean_2LNNAT = mean(abs(error));
```

```
figure(3)
plot(t_NN,x_NN(:,1:3),angles_t,angles(:,1:3),'-.');
title("One Layer Neural Net")
angles_idx=round(t_NN/angles_t(2))+1;
error = angles(angles_idx,1:3) - x_NN(:,1:3);
mean_NN = mean(abs(error));
figure(4)
plot(t_NNAT,x_NNAT(:,1:3),angles_t,angles(:,1:3),'-.');
title("One Layer Neural Net Augmented Tuning")
angles_idx=round(t_NNAT/angles_t(2))+1;
error = angles(angles_idx,1:3) - x_NNAT(:,1:3);
mean_NNAT = mean(abs(error));
figure(5)
plot(t_PD,x_PD(:,1:3),angles_t,angles(:,1:3),'-.');
title("Classical PD Controller")
angles_idx=round(t_PD/angles_t(2))+1;
error = angles(angles_idx,1:3) - x_PD(:,1:3);
mean_PD = mean(abs(error));
figure(6)
names = categorical(["2LNN" "2LNNAT" "1LNN" "1LNNAT" "PD"]);
names = reordercats(names,["2LNN" "2LNNAT" "1LNN" "1LNNAT" "PD"]);
values = [mean(mean_2LNN) mean(mean_2LNNAT) mean(mean_NN) mean(mean_NNAT) mean(mean_PD)];
bar(names , values)
```

Discrete PD

```
function xdot1=robadaptPD(t,x)
%-------------------------------------------------------------------------
% NN control with Backprop weight tuning
%-------------------------------------------------------------------------
global angles
global angles_t
global Outs
global PD_delta_t
global tau
global PD_step_num
%Compute desired trajectory
angles_idx=round(t/angles_t(2))+1;
qd=angles(angles_idx,1:3)';
if(angles_idx > 1)
   qdp=((angles(angles_idx,1:3)-angles(angles_idx-1,1:3))/angles_t(2))';
   if(angles_idx > 2)
       qdpp=(((angles(angles_idx,1:3)-angles(angles_idx-1,1:3)) -
(angles(angles_idx-1,1:3)-angles(angles_idx-2,1:3)))/angles_t(2)^2)';
   else
       qdpp=zeros(Outs,1);
   end
else
   qdp=zeros(Outs,1);
   qdpp=zeros(Outs,1);
end
q=[x(1) x(2) x(3)]';
qp=[x(4) x(5) x(6)]';
if( t > PD_delta_t*PD_step_num)
   %Adaptive control input
   Kv=20*eye(Outs); lam=5*eye(Outs); % controller parameters

   %tracking errors
   e=qd-q;
   ep=qdp-qp;
   r=ep+lam*e;
```

```
   %control torques. Parameter estimates are [x(5) x(6)]'
   tau=Kv*r;
   PD_step_num = PD_step_num +1;
end
% Robot Arm Dynamics
% m1 = weight of first link, m2 = weight of second link, m3 = payload
% weight, a1 = length of first link, a2 = length of second link
m1=0.8; m2=2.3; m3=2.7; a1=1; a2=1; g=9.8; % arm parameters
if t > 14
m3 = 2.7+1.5;
end
%Inertia M(q)
M11=(m3+m2+0.25*m1)*a1^2*(cos(q(2)))^2 + (m3+0.25*m2)*a2^2*(sin(q(2)+q(3)))^2 ...
    + (m3+m2)*a1*a2*cos(q(2))*sin(q(2)+q(3)) + 1;
M22=(m3+m2+0.25*m1)*a1^2 + (m3+0.25*m2)*a2^2 + (2*m3+m2)*a1*a2*sin(q(3)) + 1;
M23=(m3+0.25*m2)*a2^2 + (m3 + 0.5*m2)*a1*a2*sin(q(3)) + 1;
M32=M23;
M33=(m3+0.25*m2)*a2^2 + 1;
M=[M11 0 0; 0 M22 M23; 0 M32 M33];
% Coriolis/centripetal vector V(q,qdot)
V1=qp(1)*qp(2)*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3)))) - (m3+m2+0.25*m1)*a1^3*sin(2*q(2)) ...
   + (m3+m2)*a1*a2*cos(2*q(2)+q(3)) + qp(1)*qp(3)*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3)))) ...
   + (m3+m2)*a1*a2*cos(q(2))*cos(q(2)+q(3));
V2=qp(2)*qp(3)*(-(2*m3+m2)*a1*a2*cos(q(3))) + q(3)^2*(-(m3+0.5*m2)*a1*a2*cos(q(3))) ...
   + 0.5*qp(1)^2*((m3+m2+0.25*m1)*a1^2*sin(2*q(2)) + (m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3))) ...
   + (m3+0.5*m2)*a1*a2*cos(2*q(2)+q(3)));
V3=-0.5*qp(1)^2*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3))) + (m3 +
m2)*a1*a2*cos(q(2))*cos(q(2)+q(3))) ...
   - (0.5*qp(2)^2+qp(2)*qp(3))*(2*m3+m2)*a1*a2*cos(q(3));
V = [V1 V2 V3]';
% Gravity vector G(q)
G1=0;
G2=g*((m3+m2+0.5*m1)*a1*cos(q(2)) + (m3+0.5*m2)*a2*sin(q(2)+q(3)));
G3=g*(m3+0.5*m2)*a2*sin(q(2)+q(3));
G = [G1 G2 G3]';
% Manipulator Dynamics
qpp = M\(tau - V - G);
%state equations
xdot(1)=x(4);
xdot(2)=x(5);
xdot(3)=x(6);
xdot(4)=qpp(1);
xdot(5)=qpp(2);
xdot(6)=qpp(3);
xdot1=xdot';
```

Full Taylor Method

```
function xdot1=robadapt2LNNAT(t,x)
%-------------------------------------------------------------------------
% NN control with augmented weight tuning
%-------------------------------------------------------------------------
global angles
global angles_t
global I
global L
global Outs
global NN_delta_t
global PD_delta_t
global tau
```

```
global PD_step_num
global NN_step_num
persistent W
if isempty(W)
   W=zeros(L,Outs);
end
persistent V
if isempty(V)
   V=zeros(I,L-1);
end
%Compute desired trajectory
angles_idx=round(t/angles_t(2))+1;
qd=angles(angles_idx,1:3)';
if(angles_idx > 1)
   qdp=((angles(angles_idx,1:3)-angles(angles_idx-1,1:3))/angles_t(2))';
   if(angles_idx > 2)
       qdpp=(((angles(angles_idx,1:3)-angles(angles_idx-1,1:3)) -
(angles(angles_idx-1,1:3)-angles(angles_idx-2,1:3)))/angles_t(2)^2)';
   else
       qdpp=zeros(Outs,1);
   end
else
   qdp=zeros(Outs,1);
   qdpp=zeros(Outs,1);
end
q=[x(1) x(2) x(3)]';
qp=[x(4) x(5) x(6)]';
if( t > PD_delta_t*PD_step_num)
   %Adaptive control input
   Kz=50*eye(Outs); Kv=20*eye(Outs); lam=5*eye(Outs); F=50*eye(L); G=50*eye(I); kappa=.1;
ZB=.1;% controller parameters

   %tracking errors
   e=qd-q;
   ep=qdp-qp;
   r=ep+lam*e;

   %compute regression matrix
   %f=qdpp+lam*ep;
   NNIn(1)=1;NNIn(2:2+Outs-1)=e';NNIn(2+Outs:2+2*Outs-1)=ep';

NNIn(2+2*Outs:2+3*Outs-1)=qd';NNIn(2+3*Outs:2+4*Outs-1)=qdp';NNIn(2+4*Outs:2+5*Outs-1)=qdpp';
   XX=V'*NNIn';

   NNOut=logsig(XX);

   NNOut = [1 NNOut']';

   %control torques.
   tau=Kv*r+W'*NNOut + Kz*(norm([V zeros([I Outs]); zeros([L L-1]) W],"fro")-ZB)*r;
   PD_step_num = PD_step_num +1;
end
if( t > NN_delta_t*NN_step_num && t < NN_delta_t*(NN_step_num +1) )
   %parameter updates
   NNOut_dot=dlogsig(NNIn', NNOut);
   r_dot = lam*ep;
   NNIn_dot(1)=0;NNIn_dot(2:2+Outs-1)=ep';NNIn_dot(2+Outs:2+2*Outs-1)=zeros(size(ep'));

NNIn_dot(2+2*Outs:2+3*Outs-1)=qdp';NNIn_dot(2+3*Outs:2+4*Outs-1)=qdpp';NNIn_dot(2+4*Outs:2+5*O
uts-1)=zeros(size(qdpp'));
   NNIn_dot = V'*NNIn_dot';
   NNIn_dot = [0 ; NNIn_dot];
```

```
   T_w_dot=(F*NNOut*r'-F*(diag(NNOut)*(eye(L)-diag(NNOut)))*[ones(I,1)
V]'*NNIn'*r'-kappa*F*norm(r)*W)*NN_delta_t ...
       + ((NN_delta_t)^2/2)*((F*((NNIn_dot.*NNOut_dot)*r' + NNOut*r_dot')) +
(F*NNOut*r'-F*(diag(NNOut)*(eye(L)-diag(NNOut)))*[ones(I,1)
V]'*NNIn'*r'-kappa*F*norm(r)*W).*(-kappa*F*norm(r)*eye(size(W))));

   T_v_dot=(G*NNIn'*((diag(NNOut(2:end))*(eye(L-1)-diag(NNOut(2:end))))*W(2:end,:)*r)' -
kappa*G*norm(r)*V)*NN_delta_t ...
       +
((NN_delta_t)^2/2)*((G*NNIn'*((diag(NNOut(2:end))*(eye(L-1)-diag(NNOut(2:end))))*W(2:end,:)*r)
' - kappa*G*norm(r)*V).*(-kappa*G*norm(r)*eye(size(V))));
   NN_step_num = NN_step_num +1;
else
   T_w_dot=zeros(size(W));
   T_v_dot=zeros(size(V));
end
%parameter updates
W = W + T_w_dot;
V = V + T_v_dot;
% Robot Arm Dynamics
% m1 = weight of first link, m2 = weight of second link, m3 = payload
% weight, a1 = length of first link, a2 = length of second link
m1=0.8; m2=2.3; m3=2.7; a1=1; a2=1; g=9.8; % arm parameters
if t > 14
m3 = 2.7+1.5;
end
%Inertia M(q)
M11=(m3+m2+0.25*m1)*a1^2*(cos(q(2)))^2 + (m3+0.25*m2)*a2^2*(sin(q(2)+q(3)))^2 ...
    + (m3+m2)*a1*a2*cos(q(2))*sin(q(2)+q(3)) + 1;
M22=(m3+m2+0.25*m1)*a1^2 + (m3+0.25*m2)*a2^2 + (2*m3+m2)*a1*a2*sin(q(3)) + 1;
M23=(m3+0.25*m2)*a2^2 + (m3 + 0.5*m2)*a1*a2*sin(q(3)) + 1;
M32=M23;
M33=(m3+0.25*m2)*a2^2 + 1;
M=[M11 0 0; 0 M22 M23; 0 M32 M33];
% Coriolis/centripetal vector V(q,qdot)
H1=qp(1)*qp(2)*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3)))) - (m3+m2+0.25*m1)*a1^3*sin(2*q(2)) ...
   + (m3+m2)*a1*a2*cos(2*q(2)+q(3)) + qp(1)*qp(3)*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3)))) ...
   + (m3+m2)*a1*a2*cos(q(2))*cos(q(2)+q(3));
H2=qp(2)*qp(3)*(-(2*m3+m2)*a1*a2*cos(q(3))) + q(3)^2*(-(m3+0.5*m2)*a1*a2*cos(q(3))) ...
   + 0.5*qp(1)^2*((m3+m2+0.25*m1)*a1^2*sin(2*q(2)) + (m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3))) ...
   + (m3+0.5*m2)*a1*a2*cos(2*q(2)+q(3)));
H3=-0.5*qp(1)^2*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3))) + (m3 +
m2)*a1*a2*cos(q(2))*cos(q(2)+q(3))) ...
   - (0.5*qp(2)^2+qp(2)*qp(3))*(2*m3+m2)*a1*a2*cos(q(3));
H = [H1 H2 H3]';
% Gravity vector G(q)
G1=0;
G2=g*((m3+m2+0.5*m1)*a1*cos(q(2)) + (m3+0.5*m2)*a2*sin(q(2)+q(3)));
G3=g*(m3+0.5*m2)*a2*sin(q(2)+q(3));
G = [G1 G2 G3]';
% Manipulator Dynamics
qpp = M\(tau - H - G);
%state equations
xdot(1)=x(4);
xdot(2)=x(5);
xdot(3)=x(6);
xdot(4)=qpp(1);
xdot(5)=qpp(2);
xdot(6)=qpp(3);
xdot1=xdot';
```

```
function xdot1=robadapt2LNN(t,x)
%-------------------------------------------------------------------------
% NN control with Backprop weight tuning
%-------------------------------------------------------------------------
global angles
global angles_t
global I
global L
global Outs
global NN_delta_t
global PD_delta_t
global tau
global PD_step_num
global NN_step_num
persistent W
if isempty(W)
   W=zeros(L,Outs);
end
persistent V
if isempty(V)
   V=zeros(I,L-1);
end
%Compute desired trajectory
angles_idx=round(t/angles_t(2))+1;
qd=angles(angles_idx,1:3)';
if(angles_idx > 1)
   qdp=((angles(angles_idx,1:3)-angles(angles_idx-1,1:3))/angles_t(2))';
   if(angles_idx > 2)
       qdpp=(((angles(angles_idx,1:3)-angles(angles_idx-1,1:3)) -
(angles(angles_idx-1,1:3)-angles(angles_idx-2,1:3)))/angles_t(2)^2)';
   else
       qdpp=zeros(Outs,1);
   end
else
   qdp=zeros(Outs,1);
   qdpp=zeros(Outs,1);
end
q=[x(1) x(2) x(3)]';
qp=[x(4) x(5) x(6)]';
if( t > PD_delta_t*PD_step_num)
   %Adaptive control input
   Kv=20*eye(Outs); lam=5*eye(Outs); F=50*eye(L); G=50*eye(I); % controller parameters

   %tracking errors
   e=qd-q;
   ep=qdp-qp;
   r=ep+lam*e;

   %compute regression matrix
   %f=qdpp+lam*ep;
   NNIn(1)=1;NNIn(2:2+Outs-1)=e';NNIn(2+Outs:2+2*Outs-1)=ep';

NNIn(2+2*Outs:2+3*Outs-1)=qd';NNIn(2+3*Outs:2+4*Outs-1)=qdp';NNIn(2+4*Outs:2+5*Outs-1)=qdpp';
   XX=V'*NNIn';

   NNOut=logsig(XX);

   NNOut = [1 NNOut']';

   %control torques.
   tau=Kv*r+W'*NNOut;
   PD_step_num = PD_step_num +1;
```

```
end
if( t > NN_delta_t*NN_step_num )
   %parameter updates
   NNOut_dot=dlogsig(NNIn', NNOut);
   r_dot = lam*ep;
   NNIn_dot(1)=0;NNIn_dot(2:2+Outs-1)=ep';NNIn_dot(2+Outs:2+2*Outs-1)=zeros(size(ep'));

NNIn_dot(2+2*Outs:2+3*Outs-1)=qdp';NNIn_dot(2+3*Outs:2+4*Outs-1)=qdpp';NNIn_dot(2+4*Outs:2+5*O
uts-1)=zeros(size(qdpp'));
   NNIn_dot = V'*NNIn_dot';
   NNIn_dot = [0 ; NNIn_dot];
   T_w_dot=(F*NNOut*r')*NN_delta_t + ((NN_delta_t)^2/2)*(F*((NNIn_dot.*NNOut_dot)*r' +
NNOut*r_dot'));


T_v_dot=(G*NNIn'*((diag(NNOut(2:end))*(eye(L-1)-diag(NNOut(2:end))))*W(2:end,:)*r)')*NN_delta_
t;
   NN_step_num = NN_step_num +1;
else
   T_w_dot=zeros(size(W));
   T_v_dot=zeros(size(V));
end
%parameter updates
W = W + T_w_dot;
V = V + T_v_dot;
% Robot Arm Dynamics
% m1 = weight of first link, m2 = weight of second link, m3 = payload
% weight, a1 = length of first link, a2 = length of second link
m1=0.8; m2=2.3; m3=2.7; a1=1; a2=1; g=9.8; % arm parameters
if t > 14
m3 = 2.7+1.5;
end
%Inertia M(q)
M11=(m3+m2+0.25*m1)*a1^2*(cos(q(2)))^2 + (m3+0.25*m2)*a2^2*(sin(q(2)+q(3)))^2 ...
    + (m3+m2)*a1*a2*cos(q(2))*sin(q(2)+q(3)) + 1;
M22=(m3+m2+0.25*m1)*a1^2 + (m3+0.25*m2)*a2^2 + (2*m3+m2)*a1*a2*sin(q(3)) + 1;
M23=(m3+0.25*m2)*a2^2 + (m3 + 0.5*m2)*a1*a2*sin(q(3)) + 1;
M32=M23;
M33=(m3+0.25*m2)*a2^2 + 1;
M=[M11 0 0; 0 M22 M23; 0 M32 M33];
% Coriolis/centripetal vector H(q,qdot)
H1=qp(1)*qp(2)*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3)))) - (m3+m2+0.25*m1)*a1^3*sin(2*q(2)) ...
   + (m3+m2)*a1*a2*cos(2*q(2)+q(3)) + qp(1)*qp(3)*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3)))) ...
   + (m3+m2)*a1*a2*cos(q(2))*cos(q(2)+q(3));
H2=qp(2)*qp(3)*(-(2*m3+m2)*a1*a2*cos(q(3))) + q(3)^2*(-(m3+0.5*m2)*a1*a2*cos(q(3))) ...
   + 0.5*qp(1)^2*((m3+m2+0.25*m1)*a1^2*sin(2*q(2)) + (m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3))) ...
   + (m3+0.5*m2)*a1*a2*cos(2*q(2)+q(3)));
H3=-0.5*qp(1)^2*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3))) + (m3 +
m2)*a1*a2*cos(q(2))*cos(q(2)+q(3))) ...
   - (0.5*qp(2)^2+qp(2)*qp(3))*(2*m3+m2)*a1*a2*cos(q(3));
H = [H1 H2 H3]';
% Gravity vector G(q)
G1=0;
G2=g*((m3+m2+0.5*m1)*a1*cos(q(2)) + (m3+0.5*m2)*a2*sin(q(2)+q(3)));
G3=g*(m3+0.5*m2)*a2*sin(q(2)+q(3));
G = [G1 G2 G3]';
% Manipulator Dynamics
qpp = M\(tau - H - G);
%state equations
xdot(1)=x(4);
xdot(2)=x(5);
xdot(3)=x(6);
```

```
xdot(4)=qpp(1);
xdot(5)=qpp(2);
xdot(6)=qpp(3);
xdot1=xdot';


function xdot1=robadaptNNAT(t,x)
%-------------------------------------------------------------------------
% NN control with augmented weight tuning
%-------------------------------------------------------------------------
global angles
global angles_t
global I
global Outs
global NN_delta_t
global PD_delta_t
global tau
global PD_step_num
global NN_step_num
persistent W
if isempty(W)
   W=zeros(I,Outs);
end
%Compute desired trajectory
angles_idx=round(t/angles_t(2))+1;
qd=angles(angles_idx,1:3)';
if(angles_idx > 1)
   qdp=((angles(angles_idx,1:3)-angles(angles_idx-1,1:3))/angles_t(2))';
   if(angles_idx > 2)
       qdpp=(((angles(angles_idx,1:3)-angles(angles_idx-1,1:3)) -
(angles(angles_idx-1,1:3)-angles(angles_idx-2,1:3)))/angles_t(2)^2)';
   else
       qdpp=zeros(Outs,1);
   end
else
   qdp=zeros(Outs,1);
   qdpp=zeros(Outs,1);
end
%Neural Net
q=[x(1) x(2) x(3)]';
qp=[x(4) x(5) x(6)]';
if( t > PD_delta_t*PD_step_num)
   %Adaptive control input
   Kv=20*eye(Outs); lam=5*eye(Outs); F=50*eye(I); kappa=0.1;% controller parameters2
   %tracking errors
   e=qd-q;
   ep=qdp-qp;
   r=ep+lam*e;

   %compute regression matrix
   %f=qdpp+lam*ep;
   NNIn(1)=1;NNIn(2:2+Outs-1)=e';NNIn(2+Outs:2+2*Outs-1)=ep';

NNIn(2+2*Outs:2+3*Outs-1)=qd';NNIn(2+3*Outs:2+4*Outs-1)=qdp';NNIn(2+4*Outs:2+5*Outs-1)=qdpp';
   %XX=V'*NNIn';

   NNOut=logsig(NNIn');
   NNOut(1) = 1;
   %control torques. Parameter estimates are [x(5) x(6)]'
   tau=Kv*r+W'*NNOut;
   PD_step_num = PD_step_num +1;
end
```

```
if( t > NN_delta_t*NN_step_num )
   %parameter updates
   NNOut_dot=dlogsig(NNIn', NNOut);
   r_dot = lam*ep;
   NNIn_dot(1)=0;NNIn_dot(2:2+Outs-1)=ep';NNIn_dot(2+Outs:2+2*Outs-1)=zeros(size(ep'));

NNIn_dot(2+2*Outs:2+3*Outs-1)=qdp';NNIn_dot(2+3*Outs:2+4*Outs-1)=qdpp';NNIn_dot(2+4*Outs:2+5*O
uts-1)=zeros(size(qdpp'));
   T_w_dot=(F*NNOut*r'-kappa*F*norm(r)*W)*NN_delta_t ...
           + ((NN_delta_t)^2/2)*( ...
           ((F*((NNIn_dot'.*NNOut_dot)*r' + NNOut*r_dot'))-kappa*F*norm(r_dot)*W)  ...
           + (F*NNOut*r'-kappa*F*norm(r)*W).*(-kappa*F*norm(r)*eye(size(W))) );

   NN_step_num = NN_step_num +1;
else
   T_w_dot=zeros(size(W));
end
%parameter updates
W = W + T_w_dot;
% Robot Arm Dynamics
% m1 = weight of first link, m2 = weight of second link, m3 = payload
% weight, a1 = length of first link, a2 = length of second link
m1=0.8; m2=2.3; m3=2.7; a1=1; a2=1; g=9.8; % arm parameters
if t > 14
m3 = 2.7+1.5;
end
%Inertia M(q)
M11=(m3+m2+0.25*m1)*a1^2*(cos(q(2)))^2 + (m3+0.25*m2)*a2^2*(sin(q(2)+q(3)))^2 ...
    + (m3+m2)*a1*a2*cos(q(2))*sin(q(2)+q(3)) + 1;
M22=(m3+m2+0.25*m1)*a1^2 + (m3+0.25*m2)*a2^2 + (2*m3+m2)*a1*a2*sin(q(3)) + 1;
M23=(m3+0.25*m2)*a2^2 + (m3 + 0.5*m2)*a1*a2*sin(q(3)) + 1;
M32=M23;
M33=(m3+0.25*m2)*a2^2 + 1;
M=[M11 0 0; 0 M22 M23; 0 M32 M33];
% Coriolis/centripetal vector V(q,qdot)
V1=qp(1)*qp(2)*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3)))) - (m3+m2+0.25*m1)*a1^3*sin(2*q(2)) ...
   + (m3+m2)*a1*a2*cos(2*q(2)+q(3)) + qp(1)*qp(3)*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3)))) ...
   + (m3+m2)*a1*a2*cos(q(2))*cos(q(2)+q(3));
V2=qp(2)*qp(3)*(-(2*m3+m2)*a1*a2*cos(q(3))) + q(3)^2*(-(m3+0.5*m2)*a1*a2*cos(q(3))) ...
   + 0.5*qp(1)^2*((m3+m2+0.25*m1)*a1^2*sin(2*q(2)) + (m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3))) ...
   + (m3+0.5*m2)*a1*a2*cos(2*q(2)+q(3)));
V3=-0.5*qp(1)^2*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3))) + (m3 +
m2)*a1*a2*cos(q(2))*cos(q(2)+q(3))) ...
   - (0.5*qp(2)^2+qp(2)*qp(3))*(2*m3+m2)*a1*a2*cos(q(3));
V = [V1 V2 V3]';
% Gravity vector G(q)
G1=0;
G2=g*((m3+m2+0.5*m1)*a1*cos(q(2)) + (m3+0.5*m2)*a2*sin(q(2)+q(3)));
G3=g*(m3+0.5*m2)*a2*sin(q(2)+q(3));
G = [G1 G2 G3]';
% Manipulator Dynamics
qpp = M\(tau - V - G);
%state equations
xdot(1)=x(4);
xdot(2)=x(5);
xdot(3)=x(6);
xdot(4)=qpp(1);
xdot(5)=qpp(2);
xdot(6)=qpp(3);
xdot1=xdot';
```

```
function xdot1=robadaptNN(t,x)
%-------------------------------------------------------------------------
% NN control with Backprop weight tuning
%-------------------------------------------------------------------------
global angles
global angles_t
global I
global Outs
global NN_delta_t
global PD_delta_t
global tau
global PD_step_num
global NN_step_num
persistent W
if isempty(W)
   W=zeros(I,Outs);
end
%Compute desired trajectory
angles_idx=round(t/angles_t(2))+1;
qd=angles(angles_idx,1:3)';
if(angles_idx > 1)
   qdp=((angles(angles_idx,1:3)-angles(angles_idx-1,1:3))/angles_t(2))';
   if(angles_idx > 2)
       qdpp=(((angles(angles_idx,1:3)-angles(angles_idx-1,1:3)) -
(angles(angles_idx-1,1:3)-angles(angles_idx-2,1:3)))/angles_t(2)^2)';
   else
       qdpp=zeros(Outs,1);
   end
else
   qdp=zeros(Outs,1);
   qdpp=zeros(Outs,1);
end
%Neural Net
q=[x(1) x(2) x(3)]';
qp=[x(4) x(5) x(6)]';
if( t > PD_delta_t*PD_step_num)
   %Adaptive control input
   Kv=20*eye(Outs); lam=5*eye(Outs); F=50*eye(I); % controller parameters

   %tracking errors
   e=qd-q;
   ep=qdp-qp;
   r=ep+lam*e;

   %compute regression matrix
   %f=qdpp+lam*ep;
   NNIn(1)=1;NNIn(2:2+Outs-1)=e';NNIn(2+Outs:2+2*Outs-1)=ep';

NNIn(2+2*Outs:2+3*Outs-1)=qd';NNIn(2+3*Outs:2+4*Outs-1)=qdp';NNIn(2+4*Outs:2+5*Outs-1)=qdpp';
   %XX=V'*NNIn';

   NNOut=logsig(NNIn');
   NNOut(1) = 1;

   %control torques. Parameter estimates are [x(5) x(6)]'
   tau=Kv*r+W'*NNOut;
   PD_step_num = PD_step_num +1;
end
if( t > NN_delta_t*NN_step_num )
   %parameter updates
   NNOut_dot=dlogsig(NNIn', NNOut);
   r_dot = lam*ep;
```

```
   NNIn_dot(1)=0;NNIn_dot(2:2+Outs-1)=ep';NNIn_dot(2+Outs:2+2*Outs-1)=zeros(size(ep'));

NNIn_dot(2+2*Outs:2+3*Outs-1)=qdp';NNIn_dot(2+3*Outs:2+4*Outs-1)=qdpp';NNIn_dot(2+4*Outs:2+5*O
uts-1)=zeros(size(qdpp'));
   T_w_dot=(F*NNOut*r')*NN_delta_t + ((NN_delta_t)^2/2)*(F*((NNIn_dot'.*NNOut_dot)*r' +
NNOut*r_dot'));
   NN_step_num = NN_step_num +1;
else
   T_w_dot=zeros(size(W));
end
%parameter updates
W = W + T_w_dot;
% Robot Arm Dynamics
% m1 = weight of first link, m2 = weight of second link, m3 = payload
% weight, a1 = length of first link, a2 = length of second link
m1=0.8; m2=2.3; m3=2.7; a1=1; a2=1; g=9.8; % arm parameters
if t > 14
m3 = 2.7+1.5;
end
%Inertia M(q)
M11=(m3+m2+0.25*m1)*a1^2*(cos(q(2)))^2 + (m3+0.25*m2)*a2^2*(sin(q(2)+q(3)))^2 ...
    + (m3+m2)*a1*a2*cos(q(2))*sin(q(2)+q(3)) + 1;
M22=(m3+m2+0.25*m1)*a1^2 + (m3+0.25*m2)*a2^2 + (2*m3+m2)*a1*a2*sin(q(3)) + 1;
M23=(m3+0.25*m2)*a2^2 + (m3 + 0.5*m2)*a1*a2*sin(q(3)) + 1;
M32=M23;
M33=(m3+0.25*m2)*a2^2 + 1;
M=[M11 0 0; 0 M22 M23; 0 M32 M33];
% Coriolis/centripetal vector V(q,qdot)
V1=qp(1)*qp(2)*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3)))) - (m3+m2+0.25*m1)*a1^3*sin(2*q(2)) ...
   + (m3+m2)*a1*a2*cos(2*q(2)+q(3)) + qp(1)*qp(3)*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3)))) ...
   + (m3+m2)*a1*a2*cos(q(2))*cos(q(2)+q(3));
V2=qp(2)*qp(3)*(-(2*m3+m2)*a1*a2*cos(q(3))) + q(3)^2*(-(m3+0.5*m2)*a1*a2*cos(q(3))) ...
   + 0.5*qp(1)^2*((m3+m2+0.25*m1)*a1^2*sin(2*q(2)) + (m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3))) ...
   + (m3+0.5*m2)*a1*a2*cos(2*q(2)+q(3)));
V3=-0.5*qp(1)^2*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3))) + (m3 +
m2)*a1*a2*cos(q(2))*cos(q(2)+q(3))) ...
   - (0.5*qp(2)^2+qp(2)*qp(3))*(2*m3+m2)*a1*a2*cos(q(3));
V = [V1 V2 V3]';
% Gravity vector G(q)
G1=0;
G2=g*((m3+m2+0.5*m1)*a1*cos(q(2)) + (m3+0.5*m2)*a2*sin(q(2)+q(3)));
G3=g*(m3+0.5*m2)*a2*sin(q(2)+q(3));
G = [G1 G2 G3]';
% Manipulator Dynamics
qpp = M\(tau - V - G);
%state equations
xdot(1)=x(4);
xdot(2)=x(5);
xdot(3)=x(6);
xdot(4)=qpp(1);
xdot(5)=qpp(2);
xdot(6)=qpp(3);
xdot1=xdot';
```

Partial Taylor Method

```
function xdot1=robadaptNNAT(t,x)
%-------------------------------------------------------------------------
% NN control with augmented weight tuning
%-------------------------------------------------------------------------
global angles
```

```
global angles_t
global I
global Outs
global NN_delta_t
global PD_delta_t
global tau
global PD_step_num
global NN_step_num
persistent W
if isempty(W)
   W=zeros(I,Outs);
end
%Compute desired trajectory
angles_idx=round(t/angles_t(2))+1;
qd=angles(angles_idx,1:3)';
if(angles_idx > 1)
   qdp=((angles(angles_idx,1:3)-angles(angles_idx-1,1:3))/angles_t(2))';
   if(angles_idx > 2)
       qdpp=(((angles(angles_idx,1:3)-angles(angles_idx-1,1:3)) -
(angles(angles_idx-1,1:3)-angles(angles_idx-2,1:3)))/angles_t(2)^2)';
   else
       qdpp=zeros(Outs,1);
   end
else
   qdp=zeros(Outs,1);
   qdpp=zeros(Outs,1);
end
%Neural Net
q=[x(1) x(2) x(3)]';
qp=[x(4) x(5) x(6)]';
if( t > PD_delta_t*PD_step_num)
   %Adaptive control input
   Kv=20*eye(Outs); lam=5*eye(Outs); F=50*eye(I); kappa=0.1;% controller parameters2
   %tracking errors
   e=qd-q;
   ep=qdp-qp;
   r=ep+lam*e;

   %compute regression matrix
   %f=qdpp+lam*ep;
   NNIn(1)=1;NNIn(2:2+Outs-1)=e';NNIn(2+Outs:2+2*Outs-1)=ep';

NNIn(2+2*Outs:2+3*Outs-1)=qd';NNIn(2+3*Outs:2+4*Outs-1)=qdp';NNIn(2+4*Outs:2+5*Outs-1)=qdpp';
   %XX=V'*NNIn';

   NNOut=logsig(NNIn');
   NNOut(1) = 1;
   %control torques. Parameter estimates are [x(5) x(6)]'
   tau=Kv*r+W'*NNOut;
   PD_step_num = PD_step_num +1;
end
if( t > NN_delta_t*NN_step_num )
   %parameter updates
   T_w_dot=(F*NNOut*r'-kappa*F*norm(r)*W)*NN_delta_t ...
           +
((NN_delta_t)^2/2)*((F*NNOut*r'-kappa*F*norm(r)*W).*(-kappa*F*norm(r)*eye(size(W))));

   NN_step_num = NN_step_num +1;
else
   T_w_dot=zeros(size(W));
end
%parameter updates
```

```
W = W + T_w_dot;
% Robot Arm Dynamics
% m1 = weight of first link, m2 = weight of second link, m3 = payload
% weight, a1 = length of first link, a2 = length of second link
m1=0.8; m2=2.3; m3=2.7; a1=1; a2=1; g=9.8; % arm parameters
if t > 14
m3 = 2.7+1.5;
end
%Inertia M(q)
M11=(m3+m2+0.25*m1)*a1^2*(cos(q(2)))^2 + (m3+0.25*m2)*a2^2*(sin(q(2)+q(3)))^2 ...
    + (m3+m2)*a1*a2*cos(q(2))*sin(q(2)+q(3)) + 1;
M22=(m3+m2+0.25*m1)*a1^2 + (m3+0.25*m2)*a2^2 + (2*m3+m2)*a1*a2*sin(q(3)) + 1;
M23=(m3+0.25*m2)*a2^2 + (m3 + 0.5*m2)*a1*a2*sin(q(3)) + 1;
M32=M23;
M33=(m3+0.25*m2)*a2^2 + 1;
M=[M11 0 0; 0 M22 M23; 0 M32 M33];
% Coriolis/centripetal vector V(q,qdot)
V1=qp(1)*qp(2)*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3)))) - (m3+m2+0.25*m1)*a1^3*sin(2*q(2)) ...
   + (m3+m2)*a1*a2*cos(2*q(2)+q(3)) + qp(1)*qp(3)*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3)))) ...
   + (m3+m2)*a1*a2*cos(q(2))*cos(q(2)+q(3));
V2=qp(2)*qp(3)*(-(2*m3+m2)*a1*a2*cos(q(3))) + q(3)^2*(-(m3+0.5*m2)*a1*a2*cos(q(3))) ...
   + 0.5*qp(1)^2*((m3+m2+0.25*m1)*a1^2*sin(2*q(2)) + (m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3))) ...
   + (m3+0.5*m2)*a1*a2*cos(2*q(2)+q(3)));
V3=-0.5*qp(1)^2*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3))) + (m3 +
m2)*a1*a2*cos(q(2))*cos(q(2)+q(3))) ...
   - (0.5*qp(2)^2+qp(2)*qp(3))*(2*m3+m2)*a1*a2*cos(q(3));
V = [V1 V2 V3]';
% Gravity vector G(q)
G1=0;
G2=g*((m3+m2+0.5*m1)*a1*cos(q(2)) + (m3+0.5*m2)*a2*sin(q(2)+q(3)));
G3=g*(m3+0.5*m2)*a2*sin(q(2)+q(3));
G = [G1 G2 G3]';
% Manipulator Dynamics
qpp = M\(tau - V - G);
%state equations
xdot(1)=x(4);
xdot(2)=x(5);
xdot(3)=x(6);
xdot(4)=qpp(1);
xdot(5)=qpp(2);
xdot(6)=qpp(3);
xdot1=xdot';


function xdot1=robadaptNN(t,x)
%-------------------------------------------------------------------------
% NN control with Backprop weight tuning
%-------------------------------------------------------------------------
global angles
global angles_t
global I
global Outs
global NN_delta_t
global PD_delta_t
global tau
global PD_step_num
global NN_step_num
persistent W
if isempty(W)
   W=zeros(I,Outs);
end
%Compute desired trajectory
```

```
angles_idx=round(t/angles_t(2))+1;
qd=angles(angles_idx,1:3)';
if(angles_idx > 1)
   qdp=((angles(angles_idx,1:3)-angles(angles_idx-1,1:3))/angles_t(2))';
   if(angles_idx > 2)
       qdpp=(((angles(angles_idx,1:3)-angles(angles_idx-1,1:3)) -
(angles(angles_idx-1,1:3)-angles(angles_idx-2,1:3)))/angles_t(2)^2)';
   else
       qdpp=zeros(Outs,1);
   end
else
   qdp=zeros(Outs,1);
   qdpp=zeros(Outs,1);
end
%Neural Net
q=[x(1) x(2) x(3)]';
qp=[x(4) x(5) x(6)]';
if( t > PD_delta_t*PD_step_num)
   %Adaptive control input
   Kv=20*eye(Outs); lam=5*eye(Outs); F=50*eye(I); % controller parameters

   %tracking errors
   e=qd-q;
   ep=qdp-qp;
   r=ep+lam*e;

   %compute regression matrix
   %f=qdpp+lam*ep;
   NNIn(1)=1;NNIn(2:2+Outs-1)=e';NNIn(2+Outs:2+2*Outs-1)=ep';

NNIn(2+2*Outs:2+3*Outs-1)=qd';NNIn(2+3*Outs:2+4*Outs-1)=qdp';NNIn(2+4*Outs:2+5*Outs-1)=qdpp';
   %XX=V'*NNIn';

   NNOut=logsig(NNIn');
   NNOut(1) = 1;

   %control torques. Parameter estimates are [x(5) x(6)]'
   tau=Kv*r+W'*NNOut;
   PD_step_num = PD_step_num +1;
end
if( t > NN_delta_t*NN_step_num )
   %parameter updates
   T_w_dot=(F*NNOut*r')*NN_delta_t;
   NN_step_num = NN_step_num +1;
else
   T_w_dot=zeros(size(W));
end
%parameter updates
W = W + T_w_dot;
% Robot Arm Dynamics
% m1 = weight of first link, m2 = weight of second link, m3 = payload
% weight, a1 = length of first link, a2 = length of second link
m1=0.8; m2=2.3; m3=2.7; a1=1; a2=1; g=9.8; % arm parameters
if t > 14
m3 = 2.7+1.5;
end
%Inertia M(q)
M11=(m3+m2+0.25*m1)*a1^2*(cos(q(2)))^2 + (m3+0.25*m2)*a2^2*(sin(q(2)+q(3)))^2 ...
    + (m3+m2)*a1*a2*cos(q(2))*sin(q(2)+q(3)) + 1;
M22=(m3+m2+0.25*m1)*a1^2 + (m3+0.25*m2)*a2^2 + (2*m3+m2)*a1*a2*sin(q(3)) + 1;
M23=(m3+0.25*m2)*a2^2 + (m3 + 0.5*m2)*a1*a2*sin(q(3)) + 1;
M32=M23;
```

```
M33=(m3+0.25*m2)*a2^2 + 1;
M=[M11 0 0; 0 M22 M23; 0 M32 M33];
% Coriolis/centripetal vector V(q,qdot)
V1=qp(1)*qp(2)*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3)))) - (m3+m2+0.25*m1)*a1^3*sin(2*q(2)) ...
   + (m3+m2)*a1*a2*cos(2*q(2)+q(3)) + qp(1)*qp(3)*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3)))) ...
   + (m3+m2)*a1*a2*cos(q(2))*cos(q(2)+q(3));
V2=qp(2)*qp(3)*(-(2*m3+m2)*a1*a2*cos(q(3))) + q(3)^2*(-(m3+0.5*m2)*a1*a2*cos(q(3))) ...
   + 0.5*qp(1)^2*((m3+m2+0.25*m1)*a1^2*sin(2*q(2)) + (m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3))) ...
   + (m3+0.5*m2)*a1*a2*cos(2*q(2)+q(3)));
V3=-0.5*qp(1)^2*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3))) + (m3 +
m2)*a1*a2*cos(q(2))*cos(q(2)+q(3))) ...
   - (0.5*qp(2)^2+qp(2)*qp(3))*(2*m3+m2)*a1*a2*cos(q(3));
V = [V1 V2 V3]';
% Gravity vector G(q)
G1=0;
G2=g*((m3+m2+0.5*m1)*a1*cos(q(2)) + (m3+0.5*m2)*a2*sin(q(2)+q(3)));
G3=g*(m3+0.5*m2)*a2*sin(q(2)+q(3));
G = [G1 G2 G3]';
% Manipulator Dynamics
qpp = M\(tau - V - G);
%state equations
xdot(1)=x(4);
xdot(2)=x(5);
xdot(3)=x(6);
xdot(4)=qpp(1);
xdot(5)=qpp(2);
xdot(6)=qpp(3);
xdot1=xdot';


function xdot1=robadapt2LNNAT(t,x)
%-------------------------------------------------------------------------
% NN control with augmented weight tuning
%-------------------------------------------------------------------------
global angles
global angles_t
global I
global L
global Outs
global NN_delta_t
global PD_delta_t
global tau
global PD_step_num
global NN_step_num
persistent W
if isempty(W)
   W=zeros(L,Outs);
end
persistent V
if isempty(V)
   V=zeros(I,L-1);
end
%Compute desired trajectory
angles_idx=round(t/angles_t(2))+1;
qd=angles(angles_idx,1:3)';
if(angles_idx > 1)
   qdp=((angles(angles_idx,1:3)-angles(angles_idx-1,1:3))/angles_t(2))';
   if(angles_idx > 2)
       qdpp=(((angles(angles_idx,1:3)-angles(angles_idx-1,1:3)) -
(angles(angles_idx-1,1:3)-angles(angles_idx-2,1:3)))/angles_t(2)^2)';
   else
       qdpp=zeros(Outs,1);
```

```
   end
else
   qdp=zeros(Outs,1);
   qdpp=zeros(Outs,1);
end
q=[x(1) x(2) x(3)]';
qp=[x(4) x(5) x(6)]';
if( t > PD_delta_t*PD_step_num)
   %Adaptive control input
   Kz=50*eye(Outs); Kv=20*eye(Outs); lam=5*eye(Outs); F=50*eye(L); G=50*eye(I); kappa=.1;
ZB=.1;% controller parameters

   %tracking errors
   e=qd-q;
   ep=qdp-qp;
   r=ep+lam*e;

   %compute regression matrix
   %f=qdpp+lam*ep;
   NNIn(1)=1;NNIn(2:2+Outs-1)=e';NNIn(2+Outs:2+2*Outs-1)=ep';

NNIn(2+2*Outs:2+3*Outs-1)=qd';NNIn(2+3*Outs:2+4*Outs-1)=qdp';NNIn(2+4*Outs:2+5*Outs-1)=qdpp';
   XX=V'*NNIn';

   NNOut=logsig(XX);

   NNOut = [1 NNOut']';

   %control torques.
   tau=Kv*r+W'*NNOut + Kz*(norm([V zeros([I Outs]); zeros([L L-1]) W],"fro")-ZB)*r;
   PD_step_num = PD_step_num +1;
end
if( t > NN_delta_t*NN_step_num && t < NN_delta_t*(NN_step_num +1) )
   %parameter updates
   T_w_dot=(F*NNOut*r'-F*(diag(NNOut)*(eye(L)-diag(NNOut)))*[ones(I,1)
V]'*NNIn'*r'-kappa*F*norm(r)*W)*NN_delta_t ...
       + ((NN_delta_t)^2/2)*((F*NNOut*r'-F*(diag(NNOut)*(eye(L)-diag(NNOut)))*[ones(I,1)
V]'*NNIn'*r'-kappa*F*norm(r)*W).*(-kappa*F*norm(r)*eye(size(W))));

   T_v_dot=(G*NNIn'*((diag(NNOut(2:end))*(eye(L-1)-diag(NNOut(2:end))))*W(2:end,:)*r)' -
kappa*G*norm(r)*V)*NN_delta_t ...
       +
((NN_delta_t)^2/2)*((G*NNIn'*((diag(NNOut(2:end))*(eye(L-1)-diag(NNOut(2:end))))*W(2:end,:)*r)
' - kappa*G*norm(r)*V).*(-kappa*G*norm(r)*eye(size(V))));
   NN_step_num = NN_step_num +1;
else
   T_w_dot=zeros(size(W));
   T_v_dot=zeros(size(V));
end
%parameter updates
W = W + T_w_dot;
V = V + T_v_dot;
% Robot Arm Dynamics
% m1 = weight of first link, m2 = weight of second link, m3 = payload
% weight, a1 = length of first link, a2 = length of second link
m1=0.8; m2=2.3; m3=2.7; a1=1; a2=1; g=9.8; % arm parameters
if t > 14
m3 = 2.7+1.5;
end
%Inertia M(q)
M11=(m3+m2+0.25*m1)*a1^2*(cos(q(2)))^2 + (m3+0.25*m2)*a2^2*(sin(q(2)+q(3)))^2 ...
    + (m3+m2)*a1*a2*cos(q(2))*sin(q(2)+q(3)) + 1;
```

```
M22=(m3+m2+0.25*m1)*a1^2 + (m3+0.25*m2)*a2^2 + (2*m3+m2)*a1*a2*sin(q(3)) + 1;
M23=(m3+0.25*m2)*a2^2 + (m3 + 0.5*m2)*a1*a2*sin(q(3)) + 1;
M32=M23;
M33=(m3+0.25*m2)*a2^2 + 1;
M=[M11 0 0; 0 M22 M23; 0 M32 M33];
% Coriolis/centripetal vector V(q,qdot)
H1=qp(1)*qp(2)*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3)))) - (m3+m2+0.25*m1)*a1^3*sin(2*q(2)) ...
   + (m3+m2)*a1*a2*cos(2*q(2)+q(3)) + qp(1)*qp(3)*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3)))) ...
   + (m3+m2)*a1*a2*cos(q(2))*cos(q(2)+q(3));
H2=qp(2)*qp(3)*(-(2*m3+m2)*a1*a2*cos(q(3))) + q(3)^2*(-(m3+0.5*m2)*a1*a2*cos(q(3))) ...
   + 0.5*qp(1)^2*((m3+m2+0.25*m1)*a1^2*sin(2*q(2)) + (m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3))) ...
   + (m3+0.5*m2)*a1*a2*cos(2*q(2)+q(3)));
H3=-0.5*qp(1)^2*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3))) + (m3 +
m2)*a1*a2*cos(q(2))*cos(q(2)+q(3))) ...
   - (0.5*qp(2)^2+qp(2)*qp(3))*(2*m3+m2)*a1*a2*cos(q(3));
H = [H1 H2 H3]';
% Gravity vector G(q)
G1=0;
G2=g*((m3+m2+0.5*m1)*a1*cos(q(2)) + (m3+0.5*m2)*a2*sin(q(2)+q(3)));
G3=g*(m3+0.5*m2)*a2*sin(q(2)+q(3));
G = [G1 G2 G3]';
% Manipulator Dynamics
qpp = M\(tau - H - G);
%state equations
xdot(1)=x(4);
xdot(2)=x(5);
xdot(3)=x(6);
xdot(4)=qpp(1);
xdot(5)=qpp(2);
xdot(6)=qpp(3);
xdot1=xdot';


function xdot1=robadapt2LNN(t,x)
%-------------------------------------------------------------------------
% NN control with Backprop weight tuning
%-------------------------------------------------------------------------
global angles
global angles_t
global I
global L
global Outs
global NN_delta_t
global PD_delta_t
global tau
global PD_step_num
global NN_step_num
persistent W
if isempty(W)
   W=zeros(L,Outs);
end
persistent V
if isempty(V)
   V=zeros(I,L-1);
end
%Compute desired trajectory
angles_idx=round(t/angles_t(2))+1;
qd=angles(angles_idx,1:3)';
if(angles_idx > 1)
   qdp=((angles(angles_idx,1:3)-angles(angles_idx-1,1:3))/angles_t(2))';
   if(angles_idx > 2)
```

```
       qdpp=(((angles(angles_idx,1:3)-angles(angles_idx-1,1:3)) -
(angles(angles_idx-1,1:3)-angles(angles_idx-2,1:3)))/angles_t(2)^2)';
   else
       qdpp=zeros(Outs,1);
   end
else
   qdp=zeros(Outs,1);
   qdpp=zeros(Outs,1);
end
q=[x(1) x(2) x(3)]';
qp=[x(4) x(5) x(6)]';
if( t > PD_delta_t*PD_step_num)
   %Adaptive control input
   Kv=20*eye(Outs); lam=5*eye(Outs); F=50*eye(L); G=50*eye(I); % controller parameters

   %tracking errors
   e=qd-q;
   ep=qdp-qp;
   r=ep+lam*e;

   %compute regression matrix
   %f=qdpp+lam*ep;
   NNIn(1)=1;NNIn(2:2+Outs-1)=e';NNIn(2+Outs:2+2*Outs-1)=ep';

NNIn(2+2*Outs:2+3*Outs-1)=qd';NNIn(2+3*Outs:2+4*Outs-1)=qdp';NNIn(2+4*Outs:2+5*Outs-1)=qdpp';
   XX=V'*NNIn';

   NNOut=logsig(XX);

   NNOut = [1 NNOut']';

   %control torques.
   tau=Kv*r+W'*NNOut;
   PD_step_num = PD_step_num +1;
end
if( t > NN_delta_t*NN_step_num )
   %parameter updates
   T_w_dot=(F*NNOut*r')*NN_delta_t;


T_v_dot=(G*NNIn'*((diag(NNOut(2:end))*(eye(L-1)-diag(NNOut(2:end))))*W(2:end,:)*r)')*NN_delta_
t;
   NN_step_num = NN_step_num +1;
else
   T_w_dot=zeros(size(W));
   T_v_dot=zeros(size(V));
end
%parameter updates
W = W + T_w_dot;
V = V + T_v_dot;
% Robot Arm Dynamics
% m1 = weight of first link, m2 = weight of second link, m3 = payload
% weight, a1 = length of first link, a2 = length of second link
m1=0.8; m2=2.3; m3=2.7; a1=1; a2=1; g=9.8; % arm parameters
if t > 14
m3 = 2.7+1.5;
end
%Inertia M(q)
M11=(m3+m2+0.25*m1)*a1^2*(cos(q(2)))^2 + (m3+0.25*m2)*a2^2*(sin(q(2)+q(3)))^2 ...
    + (m3+m2)*a1*a2*cos(q(2))*sin(q(2)+q(3)) + 1;
M22=(m3+m2+0.25*m1)*a1^2 + (m3+0.25*m2)*a2^2 + (2*m3+m2)*a1*a2*sin(q(3)) + 1;
M23=(m3+0.25*m2)*a2^2 + (m3 + 0.5*m2)*a1*a2*sin(q(3)) + 1;
```

```
M32=M23;
M33=(m3+0.25*m2)*a2^2 + 1;
M=[M11 0 0; 0 M22 M23; 0 M32 M33];
% Coriolis/centripetal vector H(q,qdot)
H1=qp(1)*qp(2)*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3)))) - (m3+m2+0.25*m1)*a1^3*sin(2*q(2)) ...
   + (m3+m2)*a1*a2*cos(2*q(2)+q(3)) + qp(1)*qp(3)*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3)))) ...
   + (m3+m2)*a1*a2*cos(q(2))*cos(q(2)+q(3));
H2=qp(2)*qp(3)*(-(2*m3+m2)*a1*a2*cos(q(3))) + q(3)^2*(-(m3+0.5*m2)*a1*a2*cos(q(3))) ...
   + 0.5*qp(1)^2*((m3+m2+0.25*m1)*a1^2*sin(2*q(2)) + (m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3))) ...
   + (m3+0.5*m2)*a1*a2*cos(2*q(2)+q(3)));
H3=-0.5*qp(1)^2*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3))) + (m3 +
m2)*a1*a2*cos(q(2))*cos(q(2)+q(3))) ...
   - (0.5*qp(2)^2+qp(2)*qp(3))*(2*m3+m2)*a1*a2*cos(q(3));
H = [H1 H2 H3]';
% Gravity vector G(q)
G1=0;
G2=g*((m3+m2+0.5*m1)*a1*cos(q(2)) + (m3+0.5*m2)*a2*sin(q(2)+q(3)));
G3=g*(m3+0.5*m2)*a2*sin(q(2)+q(3));
G = [G1 G2 G3]';
% Manipulator Dynamics
qpp = M\(tau - H - G);
%state equations
xdot(1)=x(4);
xdot(2)=x(5);
xdot(3)=x(6);
xdot(4)=qpp(1);
xdot(5)=qpp(2);
xdot(6)=qpp(3);
xdot1=xdot';
```

Euler Method

```
function xdot1=robadapt2LNNAT(t,x)
%------------------------------------------------------------------------
% NN control with augmented weight tuning
%------------------------------------------------------------------------
global angles
global angles_t
global I
global L
global Outs
global NN_delta_t
global PD_delta_t
global tau
global PD_step_num
global NN_step_num
persistent W
if isempty(W)
   W=zeros(L,Outs);
end
persistent V
if isempty(V)
   V=zeros(I,L-1);
end
%Compute desired trajectory
angles_idx=round(t/angles_t(2))+1;
qd=angles(angles_idx,1:3)';
if(angles_idx > 1)
   qdp=((angles(angles_idx,1:3)-angles(angles_idx-1,1:3))/angles_t(2))';
   if(angles_idx > 2)
```

```
       qdpp=(((angles(angles_idx,1:3)-angles(angles_idx-1,1:3)) -
(angles(angles_idx-1,1:3)-angles(angles_idx-2,1:3)))/angles_t(2)^2)';
   else
       qdpp=zeros(Outs,1);
   end
else
   qdp=zeros(Outs,1);
   qdpp=zeros(Outs,1);
end
q=[x(1) x(2) x(3)]';
qp=[x(4) x(5) x(6)]';
if( t > PD_delta_t*PD_step_num)
   %Adaptive control input
   Kz=50*eye(Outs); Kv=20*eye(Outs); lam=5*eye(Outs); F=50*eye(L); G=50*eye(I); kappa=.1;
ZB=.1;% controller parameters

   %tracking errors
   e=qd-q;
   ep=qdp-qp;
   r=ep+lam*e;

   %compute regression matrix
   %f=qdpp+lam*ep;
   NNIn(1)=1;NNIn(2:2+Outs-1)=e';NNIn(2+Outs:2+2*Outs-1)=ep';

NNIn(2+2*Outs:2+3*Outs-1)=qd';NNIn(2+3*Outs:2+4*Outs-1)=qdp';NNIn(2+4*Outs:2+5*Outs-1)=qdpp';
   XX=V'*NNIn';

   NNOut=logsig(XX);

   NNOut = [1 NNOut']';

   %control torques.
   tau=Kv*r+W'*NNOut + Kz*(norm([V zeros([I Outs]); zeros([L L-1]) W],"fro")-ZB)*r;
   PD_step_num = PD_step_num +1;
end
if( t > NN_delta_t*NN_step_num && t < NN_delta_t*(NN_step_num +1) )
   T_w_dot=(F*NNOut*r'-F*(diag(NNOut)*(eye(L)-diag(NNOut)))*[ones(I,1)
V]'*NNIn'*r'-kappa*F*norm(r)*W)*NN_delta_t;

   T_v_dot=(G*NNIn'*((diag(NNOut(2:end))*(eye(L-1)-diag(NNOut(2:end))))*W(2:end,:)*r)' -
kappa*G*norm(r)*V)*NN_delta_t;
   NN_step_num = NN_step_num +1;
else
   T_w_dot=zeros(size(W));
   T_v_dot=zeros(size(V));
end
%parameter updates
W = W + T_w_dot;
V = V + T_v_dot;
% Robot Arm Dynamics
% m1 = weight of first link, m2 = weight of second link, m3 = payload
% weight, a1 = length of first link, a2 = length of second link
m1=0.8; m2=2.3; m3=2.7; a1=1; a2=1; g=9.8; % arm parameters
if t > 14
m3 = 2.7+1.5;
end
%Inertia M(q)
M11=(m3+m2+0.25*m1)*a1^2*(cos(q(2)))^2 + (m3+0.25*m2)*a2^2*(sin(q(2)+q(3)))^2 ...
    + (m3+m2)*a1*a2*cos(q(2))*sin(q(2)+q(3)) + 1;
M22=(m3+m2+0.25*m1)*a1^2 + (m3+0.25*m2)*a2^2 + (2*m3+m2)*a1*a2*sin(q(3)) + 1;
M23=(m3+0.25*m2)*a2^2 + (m3 + 0.5*m2)*a1*a2*sin(q(3)) + 1;
```

```
M32=M23;
M33=(m3+0.25*m2)*a2^2 + 1;
M=[M11 0 0; 0 M22 M23; 0 M32 M33];
% Coriolis/centripetal vector V(q,qdot)
H1=qp(1)*qp(2)*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3)))) - (m3+m2+0.25*m1)*a1^3*sin(2*q(2)) ...
   + (m3+m2)*a1*a2*cos(2*q(2)+q(3)) + qp(1)*qp(3)*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3)))) ...
   + (m3+m2)*a1*a2*cos(q(2))*cos(q(2)+q(3));
H2=qp(2)*qp(3)*(-(2*m3+m2)*a1*a2*cos(q(3))) + q(3)^2*(-(m3+0.5*m2)*a1*a2*cos(q(3))) ...
   + 0.5*qp(1)^2*((m3+m2+0.25*m1)*a1^2*sin(2*q(2)) + (m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3))) ...
   + (m3+0.5*m2)*a1*a2*cos(2*q(2)+q(3)));
H3=-0.5*qp(1)^2*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3))) + (m3 +
m2)*a1*a2*cos(q(2))*cos(q(2)+q(3))) ...
   - (0.5*qp(2)^2+qp(2)*qp(3))*(2*m3+m2)*a1*a2*cos(q(3));
H = [H1 H2 H3]';
% Gravity vector G(q)
G1=0;
G2=g*((m3+m2+0.5*m1)*a1*cos(q(2)) + (m3+0.5*m2)*a2*sin(q(2)+q(3)));
G3=g*(m3+0.5*m2)*a2*sin(q(2)+q(3));
G = [G1 G2 G3]';
% Manipulator Dynamics
qpp = M\(tau - H - G);
%state equations
xdot(1)=x(4);
xdot(2)=x(5);
xdot(3)=x(6);
xdot(4)=qpp(1);
xdot(5)=qpp(2);
xdot(6)=qpp(3);
xdot1=xdot';


function xdot1=robadapt2LNN(t,x)
%-------------------------------------------------------------------------
% NN control with Backprop weight tuning
%-------------------------------------------------------------------------
global angles
global angles_t
global I
global L
global Outs
global NN_delta_t
global PD_delta_t
global tau
global PD_step_num
global NN_step_num
persistent W
if isempty(W)
   W=zeros(L,Outs);
end
persistent V
if isempty(V)
   V=zeros(I,L-1);
end
%Compute desired trajectory
angles_idx=round(t/angles_t(2))+1;
qd=angles(angles_idx,1:3)';
if(angles_idx > 1)
   qdp=((angles(angles_idx,1:3)-angles(angles_idx-1,1:3))/angles_t(2))';
   if(angles_idx > 2)
       qdpp=(((angles(angles_idx,1:3)-angles(angles_idx-1,1:3)) -
(angles(angles_idx-1,1:3)-angles(angles_idx-2,1:3)))/angles_t(2)^2)';
   else
```

```
        qdpp=zeros(Outs,1);
    end
else
    qdp=zeros(Outs,1);
    qdpp=zeros(Outs,1);
end
q=[x(1) x(2) x(3)]';
qp=[x(4) x(5) x(6)]';
if( t > PD_delta_t*PD_step_num)
    %Adaptive control input
    Kv=20*eye(Outs); lam=5*eye(Outs); F=50*eye(L); G=50*eye(I); % controller parameters

    %tracking errors
    e=qd-q;
    ep=qdp-qp;
    r=ep+lam*e;

    %compute regression matrix
    %f=qdpp+lam*ep;
    NNIn(1)=1;NNIn(2:2+Outs-1)=e';NNIn(2+Outs:2+2*Outs-1)=ep';

NNIn(2+2*Outs:2+3*Outs-1)=qd';NNIn(2+3*Outs:2+4*Outs-1)=qdp';NNIn(2+4*Outs:2+5*Outs-1)=qdpp';
    XX=V'*NNIn';

    NNOut=logsig(XX);

    NNOut = [1 NNOut']';

    %control torques.
    tau=Kv*r+W'*NNOut;
    PD_step_num = PD_step_num +1;
end
if( t > NN_delta_t*NN_step_num )
    %parameter updates
    T_w_dot=(F*NNOut*r')*NN_delta_t;


T_v_dot=(G*NNIn'*((diag(NNOut(2:end))*(eye(L-1)-diag(NNOut(2:end))))*W(2:end,:)*r)')*NN_delta_
t;
    NN_step_num = NN_step_num +1;
else
    T_w_dot=zeros(size(W));
    T_v_dot=zeros(size(V));
end
%parameter updates
W = W + T_w_dot;
V = V + T_v_dot;
% Robot Arm Dynamics
% m1 = weight of first link, m2 = weight of second link, m3 = payload
% weight, a1 = length of first link, a2 = length of second link
m1=0.8; m2=2.3; m3=2.7; a1=1; a2=1; g=9.8; % arm parameters
if t > 14
m3 = 2.7+1.5;
end
%Inertia M(q)
M11=(m3+m2+0.25*m1)*a1^2*(cos(q(2)))^2 + (m3+0.25*m2)*a2^2*(sin(q(2)+q(3)))^2 ...
     + (m3+m2)*a1*a2*cos(q(2))*sin(q(2)+q(3)) + 1;
M22=(m3+m2+0.25*m1)*a1^2 + (m3+0.25*m2)*a2^2 + (2*m3+m2)*a1*a2*sin(q(3)) + 1;
M23=(m3+0.25*m2)*a2^2 + (m3 + 0.5*m2)*a1*a2*sin(q(3)) + 1;
M32=M23;
M33=(m3+0.25*m2)*a2^2 + 1;
M=[M11 0 0; 0 M22 M23; 0 M32 M33];
```

```
% Coriolis/centripetal vector H(q,qdot)
H1=qp(1)*qp(2)*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3)))) - (m3+m2+0.25*m1)*a1^3*sin(2*q(2)) ...
   + (m3+m2)*a1*a2*cos(2*q(2)+q(3)) + qp(1)*qp(3)*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3)))) ...
   + (m3+m2)*a1*a2*cos(q(2))*cos(q(2)+q(3));
H2=qp(2)*qp(3)*(-(2*m3+m2)*a1*a2*cos(q(3))) + q(3)^2*(-(m3+0.5*m2)*a1*a2*cos(q(3))) ...
   + 0.5*qp(1)^2*((m3+m2+0.25*m1)*a1^2*sin(2*q(2)) + (m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3))) ...
   + (m3+0.5*m2)*a1*a2*cos(2*q(2)+q(3)));
H3=-0.5*qp(1)^2*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3))) + (m3 +
m2)*a1*a2*cos(q(2))*cos(q(2)+q(3))) ...
   - (0.5*qp(2)^2+qp(2)*qp(3))*(2*m3+m2)*a1*a2*cos(q(3));
H = [H1 H2 H3]';
% Gravity vector G(q)
G1=0;
G2=g*((m3+m2+0.5*m1)*a1*cos(q(2)) + (m3+0.5*m2)*a2*sin(q(2)+q(3)));
G3=g*(m3+0.5*m2)*a2*sin(q(2)+q(3));
G = [G1 G2 G3]';
% Manipulator Dynamics
qpp = M\(tau - H - G);
%state equations
xdot(1)=x(4);
xdot(2)=x(5);
xdot(3)=x(6);
xdot(4)=qpp(1);
xdot(5)=qpp(2);
xdot(6)=qpp(3);
xdot1=xdot';


function xdot1=robadaptNNAT(t,x)
%-------------------------------------------------------------------------
% NN control with augmented weight tuning
%-------------------------------------------------------------------------
global angles
global angles_t
global I
global Outs
global NN_delta_t
global PD_delta_t
global tau
global PD_step_num
global NN_step_num
persistent W
if isempty(W)
   W=zeros(I,Outs);
end
%Compute desired trajectory
angles_idx=round(t/angles_t(2))+1;
qd=angles(angles_idx,1:3)';
if(angles_idx > 1)
   qdp=((angles(angles_idx,1:3)-angles(angles_idx-1,1:3))/angles_t(2))';
   if(angles_idx > 2)
       qdpp=(((angles(angles_idx,1:3)-angles(angles_idx-1,1:3)) -
(angles(angles_idx-1,1:3)-angles(angles_idx-2,1:3)))/angles_t(2)^2)';
   else
       qdpp=zeros(Outs,1);
   end
else
   qdp=zeros(Outs,1);
   qdpp=zeros(Outs,1);
end
%Neural Net
q=[x(1) x(2) x(3)]';
```

```
qp=[x(4) x(5) x(6)]';
if( t > PD_delta_t*PD_step_num)
   %Adaptive control input
   Kv=20*eye(Outs); lam=5*eye(Outs); F=50*eye(I); kappa=0.1;% controller parameters2
   %tracking errors
   e=qd-q;
   ep=qdp-qp;
   r=ep+lam*e;

   %compute regression matrix
   %f=qdpp+lam*ep;
   NNIn(1)=1;NNIn(2:2+Outs-1)=e';NNIn(2+Outs:2+2*Outs-1)=ep';

NNIn(2+2*Outs:2+3*Outs-1)=qd';NNIn(2+3*Outs:2+4*Outs-1)=qdp';NNIn(2+4*Outs:2+5*Outs-1)=qdpp';
   %XX=V'*NNIn';

   NNOut=logsig(NNIn');
   NNOut(1) = 1;
   %control torques. Parameter estimates are [x(5) x(6)]'
   tau=Kv*r+W'*NNOut;
   PD_step_num = PD_step_num +1;
end
if( t > NN_delta_t*NN_step_num )
   %parameter updates
   %T_w_dot=F*NNOut*r'-kappa*F*norm(r)*W;
   T_w_dot=(F*NNOut*r'-kappa*F*norm(r)*W)*NN_delta_t;

   NN_step_num = NN_step_num +1;
else
   T_w_dot=zeros(size(W));
end
%parameter updates
W = W + T_w_dot;
% Robot Arm Dynamics
% m1 = weight of first link, m2 = weight of second link, m3 = payload
% weight, a1 = length of first link, a2 = length of second link
m1=0.8; m2=2.3; m3=2.7; a1=1; a2=1; g=9.8; % arm parameters
if t > 14
m3 = 2.7+1.5;
end
%Inertia M(q)
M11=(m3+m2+0.25*m1)*a1^2*(cos(q(2)))^2 + (m3+0.25*m2)*a2^2*(sin(q(2)+q(3)))^2 ...
    + (m3+m2)*a1*a2*cos(q(2))*sin(q(2)+q(3)) + 1;
M22=(m3+m2+0.25*m1)*a1^2 + (m3+0.25*m2)*a2^2 + (2*m3+m2)*a1*a2*sin(q(3)) + 1;
M23=(m3+0.25*m2)*a2^2 + (m3 + 0.5*m2)*a1*a2*sin(q(3)) + 1;
M32=M23;
M33=(m3+0.25*m2)*a2^2 + 1;
M=[M11 0 0; 0 M22 M23; 0 M32 M33];
% Coriolis/centripetal vector V(q,qdot)
V1=qp(1)*qp(2)*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3)))) - (m3+m2+0.25*m1)*a1^3*sin(2*q(2)) ...
   + (m3+m2)*a1*a2*cos(2*q(2)+q(3)) + qp(1)*qp(3)*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3)))) ...
   + (m3+m2)*a1*a2*cos(q(2))*cos(q(2)+q(3));
V2=qp(2)*qp(3)*(-(2*m3+m2)*a1*a2*cos(q(3))) + q(3)^2*(-(m3+0.5*m2)*a1*a2*cos(q(3))) ...
   + 0.5*qp(1)^2*((m3+m2+0.25*m1)*a1^2*sin(2*q(2)) + (m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3))) ...
   + (m3+0.5*m2)*a1*a2*cos(2*q(2)+q(3)));
V3=-0.5*qp(1)^2*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3))) + (m3 +
m2)*a1*a2*cos(q(2))*cos(q(2)+q(3))) ...
   - (0.5*qp(2)^2+qp(2)*qp(3))*(2*m3+m2)*a1*a2*cos(q(3));
V = [V1 V2 V3]';
% Gravity vector G(q)
G1=0;
G2=g*((m3+m2+0.5*m1)*a1*cos(q(2)) + (m3+0.5*m2)*a2*sin(q(2)+q(3)));
```

```
G3=g*(m3+0.5*m2)*a2*sin(q(2)+q(3));
G = [G1 G2 G3]';
% Manipulator Dynamics
qpp = M\(tau - V - G);
%state equations
xdot(1)=x(4);
xdot(2)=x(5);
xdot(3)=x(6);
xdot(4)=qpp(1);
xdot(5)=qpp(2);
xdot(6)=qpp(3);
xdot1=xdot';


function xdot1=robadaptNN(t,x)
%-------------------------------------------------------------------------
% NN control with Backprop weight tuning
%-------------------------------------------------------------------------
global angles
global angles_t
global I
global Outs
global NN_delta_t
global PD_delta_t
global tau
global PD_step_num
global NN_step_num
persistent W
if isempty(W)
   W=zeros(I,Outs);
end
%Compute desired trajectory
angles_idx=round(t/angles_t(2))+1;
qd=angles(angles_idx,1:3)';
if(angles_idx > 1)
   qdp=((angles(angles_idx,1:3)-angles(angles_idx-1,1:3))/angles_t(2))';
   if(angles_idx > 2)
       qdpp=(((angles(angles_idx,1:3)-angles(angles_idx-1,1:3)) -
(angles(angles_idx-1,1:3)-angles(angles_idx-2,1:3)))/angles_t(2)^2)';
   else
       qdpp=zeros(Outs,1);
   end
else
   qdp=zeros(Outs,1);
   qdpp=zeros(Outs,1);
end
%Neural Net
q=[x(1) x(2) x(3)]';
qp=[x(4) x(5) x(6)]';
if( t > PD_delta_t*PD_step_num)
   %Adaptive control input
   Kv=20*eye(Outs); lam=5*eye(Outs); F=50*eye(I); % controller parameters

   %tracking errors
   e=qd-q;
   ep=qdp-qp;
   r=ep+lam*e;

   %compute regression matrix
   %f=qdpp+lam*ep;
   NNIn(1)=1;NNIn(2:2+Outs-1)=e';NNIn(2+Outs:2+2*Outs-1)=ep';
```

```
NNIn(2+2*Outs:2+3*Outs-1)=qd';NNIn(2+3*Outs:2+4*Outs-1)=qdp';NNIn(2+4*Outs:2+5*Outs-1)=qdpp';
   %XX=V'*NNIn';

   NNOut=logsig(NNIn');
   NNOut(1) = 1;

   %control torques. Parameter estimates are [x(5) x(6)]'
   tau=Kv*r+W'*NNOut;
   PD_step_num = PD_step_num +1;
end
if( t > NN_delta_t*NN_step_num )
   %parameter updates
   T_w_dot=(F*NNOut*r')*NN_delta_t;
   NN_step_num = NN_step_num +1;
else
   T_w_dot=zeros(size(W));
end
%parameter updates
W = W + T_w_dot;
% Robot Arm Dynamics
% m1 = weight of first link, m2 = weight of second link, m3 = payload
% weight, a1 = length of first link, a2 = length of second link
m1=0.8; m2=2.3; m3=2.7; a1=1; a2=1; g=9.8; % arm parameters
if t > 14
m3 = 2.7+1.5;
end
%Inertia M(q)
M11=(m3+m2+0.25*m1)*a1^2*(cos(q(2)))^2 + (m3+0.25*m2)*a2^2*(sin(q(2)+q(3)))^2 ...
    + (m3+m2)*a1*a2*cos(q(2))*sin(q(2)+q(3)) + 1;
M22=(m3+m2+0.25*m1)*a1^2 + (m3+0.25*m2)*a2^2 + (2*m3+m2)*a1*a2*sin(q(3)) + 1;
M23=(m3+0.25*m2)*a2^2 + (m3 + 0.5*m2)*a1*a2*sin(q(3)) + 1;
M32=M23;
M33=(m3+0.25*m2)*a2^2 + 1;
M=[M11 0 0; 0 M22 M23; 0 M32 M33];
% Coriolis/centripetal vector V(q,qdot)
V1=qp(1)*qp(2)*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3)))) - (m3+m2+0.25*m1)*a1^3*sin(2*q(2)) ...
   + (m3+m2)*a1*a2*cos(2*q(2)+q(3)) + qp(1)*qp(3)*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3)))) ...
   + (m3+m2)*a1*a2*cos(q(2))*cos(q(2)+q(3));
V2=qp(2)*qp(3)*(-(2*m3+m2)*a1*a2*cos(q(3))) + q(3)^2*(-(m3+0.5*m2)*a1*a2*cos(q(3))) ...
   + 0.5*qp(1)^2*((m3+m2+0.25*m1)*a1^2*sin(2*q(2)) + (m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3))) ...
   + (m3+0.5*m2)*a1*a2*cos(2*q(2)+q(3)));
V3=-0.5*qp(1)^2*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3))) + (m3 +
m2)*a1*a2*cos(q(2))*cos(q(2)+q(3))) ...
   - (0.5*qp(2)^2+qp(2)*qp(3))*(2*m3+m2)*a1*a2*cos(q(3));
V = [V1 V2 V3]';
% Gravity vector G(q)
G1=0;
G2=g*((m3+m2+0.5*m1)*a1*cos(q(2)) + (m3+0.5*m2)*a2*sin(q(2)+q(3)));
G3=g*(m3+0.5*m2)*a2*sin(q(2)+q(3));
G = [G1 G2 G3]';
% Manipulator Dynamics
qpp = M\(tau - V - G);
%state equations
xdot(1)=x(4);
xdot(2)=x(5);
xdot(3)=x(6);
xdot(4)=qpp(1);
xdot(5)=qpp(2);
xdot(6)=qpp(3);
xdot1=xdot';
```

Continuous Time

```
clear all;
clc;
global angles
global angles_t
global I
global L
global Outs
tspan=[0, 20];
angles=load("joint_angles.mat");
angles=pi*angles.q/180;
angles = [angles ; angles];
angles_len=length(angles);
angles_t=0:((tspan(2)+(tspan(2)/angles_len))/angles_len):tspan(2);
I = 16; % Num of NN inputs
L=10; % Num hidden layer neurons
L=L+1;
Outs=3; % Num NN outputs
WW=zeros(L,Outs);
VV=zeros(I,L);
x0_PD = [0 0 0 0 0 0]';
x0=[x0_PD' WW(:,1)' WW(:,2)' WW(:,3)']';
x0_2L=[x0' VV(:,1)' VV(:,2)' VV(:,3)' VV(:,4)' VV(:,5)' VV(:,6)' VV(:,7)' ...
      VV(:,8)' VV(:,9)' VV(:,10)']';
WW_1L=zeros(I,Outs);
x0_1L=[x0_PD' WW_1L(:,1)' WW_1L(:,2)' WW_1L(:,3)']';
[t_2LNN,x_2LNN]=ode45('robadapt2LNN',tspan,x0_2L);
[t_2LNNAT,x_2LNNAT]=ode45('robadapt2LNNAT',tspan,x0_2L);
[t_NN,x_NN]=ode45('robadaptNN',tspan,x0_1L);
[t_NNAT,x_NNAT]=ode45('robadaptNNAT',tspan,x0_1L);
[t_PD, x_PD]=ode45('robadaptPD', tspan,x0_PD);
figure(1)
plot(t_2LNN,x_2LNN(:,1:3),angles_t,angles(:,1:3),'-.');
title("Two Layer Neural Net");
angles_idx=round(t_2LNN/angles_t(2))+1;
error = angles(angles_idx,1:3) - x_2LNN(:,1:3);
mean_2LNN = mean(abs(error));
figure(2)
plot(t_2LNNAT,x_2LNNAT(:,1:3),angles_t,angles(:,1:3),'-.');
title("Two Layer Neural Net Augmented Tuning")
angles_idx=round(t_2LNNAT/angles_t(2))+1;
error = angles(angles_idx,1:3) - x_2LNNAT(:,1:3);
mean_2LNNAT = mean(abs(error));
figure(3)
plot(t_NN,x_NN(:,1:3),angles_t,angles(:,1:3),'-.');
title("One Layer Neural Net")
angles_idx=round(t_NN/angles_t(2))+1;
error = angles(angles_idx,1:3) - x_NN(:,1:3);
mean_NN = mean(abs(error));
figure(4)
plot(t_NNAT,x_NNAT(:,1:3),angles_t,angles(:,1:3),'-.');
title("One Layer Neural Net Augmented Tuning")
angles_idx=round(t_NNAT/angles_t(2))+1;
error = angles(angles_idx,1:3) - x_NNAT(:,1:3);
mean_NNAT = mean(abs(error));
figure(5)
plot(t_PD,x_PD(:,1:3),angles_t,angles(:,1:3),'-.');
title("Classical PD Controller")
angles_idx=round(t_PD/angles_t(2))+1;
error = angles(angles_idx,1:3) - x_PD(:,1:3);
mean_PD = mean(abs(error));
```

```
figure(6)
names = categorical(["2LNN" "2LNNAT" "1LNN" "1LNNAT" "PD"]);
names = reordercats(names,["2LNN" "2LNNAT" "1LNN" "1LNNAT" "PD"]);
values = [mean(mean_2LNN) mean(mean_2LNNAT) mean(mean_NN) mean(mean_NNAT) mean(mean_PD)];
bar(names , values)
```

```
% file robout.m
function [qd,e]= robout(t,x)
%Compute desired trajectory
period=5; amp1=1; amp2=1;
fact=2*pi/period;
sinf=sin(fact*t);
cosf=cos(fact*t);
qd=[amp1*sinf amp2*cosf];
% tracking errors
   e= qd - x(:,1:2);
```

```
function xdot1=robadapt2LNNAT(t,x)
%-------------------------------------------------------------------------
% NN control with augmented weight tuning
%-------------------------------------------------------------------------
global angles
global angles_t
global I
global L
global Outs
%Compute desired trajectory
angles_idx=round(t/angles_t(2))+1;
qd=angles(angles_idx,1:3)';
if(angles_idx > 1)
   qdp=((angles(angles_idx,1:3)-angles(angles_idx-1,1:3))/angles_t(2))';
   if(angles_idx > 2)
       qdpp=(((angles(angles_idx,1:3)-angles(angles_idx-1,1:3)) -
(angles(angles_idx-1,1:3)-angles(angles_idx-2,1:3)))/angles_t(2)^2)';
   else
       qdpp=zeros(Outs,1);
   end
else
   qdp=zeros(Outs,1);
   qdpp=zeros(Outs,1);
end
%Neural Net
W_start = Outs*2 +1;
W=[x(W_start:W_start+L-1) x(W_start+L:W_start+2*L-1) x(W_start+2*L:W_start+3*L-1)];
V_start = W_start+3*L;
V=[x(V_start:V_start+I-1) x(V_start+I:V_start+2*I-1) x(V_start+2*I:V_start+3*I-1)
x(V_start+3*I:V_start+4*I-1) x(V_start+4*I:V_start+5*I-1) x(V_start+5*I:V_start+6*I-1)
x(V_start+6*I:V_start+7*I-1) x(V_start+7*I:V_start+8*I-1) x(V_start+8*I:V_start+9*I-1)
x(V_start+9*I:V_start+10*I-1)]; %ones(11,L);
q=[x(1) x(2) x(3)]';
qp=[x(4) x(5) x(6)]';
%Adaptive control input
Kz=50*eye(Outs); Kv=20*eye(Outs); lam=5*eye(Outs); F=50*eye(L); G=50*eye(I); kappa=.1; ZB=.1;%
controller parameters
%tracking errors
e=qd-q;
ep=qdp-qp;
r=ep+lam*e;
%compute regression matrix
```

```
%f=qdpp+lam*ep;
NNIn(1)=1;NNIn(2:2+Outs-1)=e';NNIn(2+Outs:2+2*Outs-1)=ep';
NNIn(2+2*Outs:2+3*Outs-1)=qd';NNIn(2+3*Outs:2+4*Outs-1)=qdp';NNIn(2+4*Outs:2+5*Outs-1)=qdpp';
XX=V'*NNIn';
NNOut=logsig(XX);
NNOut = [1 NNOut']';
%control torques.
tau=Kv*r+W'*NNOut + Kz*(norm([V zeros([I Outs]); zeros([L L-1]) W],"fro")-ZB)*r;
%parameter updates
T_w_dot=F*NNOut*r'-F*(diag(NNOut)*(eye(L)-diag(NNOut)))*[ones(I,1)
V]'*NNIn'*r'-kappa*F*norm(r)*W;
for i = 1:Outs
   xdot(W_start+(i-1)*L:W_start+i*L-1) = T_w_dot(:,i)';
end
T_v_dot=G*NNIn'*((diag(NNOut(2:end))*(eye(L-1)-diag(NNOut(2:end))))*W(2:end,:)*r)' -
kappa*G*norm(r)*V;
for i = 1:(L-1)
   xdot(V_start+(i-1)*I:V_start+i*I-1) = T_v_dot(:,i)';
end
% Robot Arm Dynamics
% m1 = weight of first link, m2 = weight of second link, m3 = payload
% weight, a1 = length of first link, a2 = length of second link
m1=0.8; m2=2.3; m3=2.7; a1=1; a2=1; g=9.8; % arm parameters
if t > 14
m3 = 2.7+1.5;
end
%Inertia M(q)
M11=(m3+m2+0.25*m1)*a1^2*(cos(q(2)))^2 + (m3+0.25*m2)*a2^2*(sin(q(2)+q(3)))^2 ...
    + (m3+m2)*a1*a2*cos(q(2))*sin(q(2)+q(3)) + 1;
M22=(m3+m2+0.25*m1)*a1^2 + (m3+0.25*m2)*a2^2 + (2*m3+m2)*a1*a2*sin(q(3)) + 1;
M23=(m3+0.25*m2)*a2^2 + (m3 + 0.5*m2)*a1*a2*sin(q(3)) + 1;
M32=M23;
M33=(m3+0.25*m2)*a2^2 + 1;
M=[M11 0 0; 0 M22 M23; 0 M32 M33];
% Coriolis/centripetal vector V(q,qdot)
V1=qp(1)*qp(2)*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3)))) - (m3+m2+0.25*m1)*a1^3*sin(2*q(2)) ...
   + (m3+m2)*a1*a2*cos(2*q(2)+q(3)) + qp(1)*qp(3)*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3)))) ...
   + (m3+m2)*a1*a2*cos(q(2))*cos(q(2)+q(3));
V2=qp(2)*qp(3)*(-(2*m3+m2)*a1*a2*cos(q(3))) + q(3)^2*(-(m3+0.5*m2)*a1*a2*cos(q(3))) ...
   + 0.5*qp(1)^2*((m3+m2+0.25*m1)*a1^2*sin(2*q(2)) + (m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3))) ...
   + (m3+0.5*m2)*a1*a2*cos(2*q(2)+q(3)));
V3=-0.5*qp(1)^2*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3))) + (m3 +
m2)*a1*a2*cos(q(2))*cos(q(2)+q(3))) ...
   - (0.5*qp(2)^2+qp(2)*qp(3))*(2*m3+m2)*a1*a2*cos(q(3));
V = [V1 V2 V3]';
% Gravity vector G(q)
G1=0;
G2=g*((m3+m2+0.5*m1)*a1*cos(q(2)) + (m3+0.5*m2)*a2*sin(q(2)+q(3)));
G3=g*(m3+0.5*m2)*a2*sin(q(2)+q(3));
G = [G1 G2 G3]';
% Manipulator Dynamics
qpp = M\(tau - V - G);
%state equations
xdot(1)=x(4);
xdot(2)=x(5);
xdot(3)=x(6);
xdot(4)=qpp(1);
xdot(5)=qpp(2);
xdot(6)=qpp(3);
xdot1=xdot';
```

```
function xdot1=robadapt2LNN(t,x)
%-------------------------------------------------------------------------
% NN control with Backprop weight tuning
%-------------------------------------------------------------------------
global angles
global angles_t
global I
global L
global Outs
%Compute desired trajectory
angles_idx=round(t/angles_t(2))+1;
qd=angles(angles_idx,1:3)';
if(angles_idx > 1)
   qdp=((angles(angles_idx,1:3)-angles(angles_idx-1,1:3))/angles_t(2))';
   if(angles_idx > 2)
       qdpp=(((angles(angles_idx,1:3)-angles(angles_idx-1,1:3)) -
(angles(angles_idx-1,1:3)-angles(angles_idx-2,1:3)))/angles_t(2)^2)';
   else
       qdpp=zeros(Outs,1);
   end
else
   qdp=zeros(Outs,1);
   qdpp=zeros(Outs,1);
end
% Neural Network Configuration
W_start = Outs*2 +1;
W=[x(W_start:W_start+L-1) x(W_start+L:W_start+2*L-1) x(W_start+2*L:W_start+3*L-1)];
V_start = W_start+3*L;
V=[x(V_start:V_start+I-1) x(V_start+I:V_start+2*I-1) x(V_start+2*I:V_start+3*I-1)
x(V_start+3*I:V_start+4*I-1) x(V_start+4*I:V_start+5*I-1) x(V_start+5*I:V_start+6*I-1)
x(V_start+6*I:V_start+7*I-1) x(V_start+7*I:V_start+8*I-1) x(V_start+8*I:V_start+9*I-1)
x(V_start+9*I:V_start+10*I-1)]; %ones(11,L);
q=[x(1) x(2) x(3)]';
qp=[x(4) x(5) x(6)]';
%Adaptive control input
Kv=20*eye(Outs); lam=5*eye(Outs); F=50*eye(L); G=50*eye(I); % controller parameters
%tracking errors
e=qd-q;
ep=qdp-qp;
r=ep+lam*e;
%compute regression matrix
%f=qdpp+lam*ep;
NNIn(1)=1;NNIn(2:2+Outs-1)=e';NNIn(2+Outs:2+2*Outs-1)=ep';
NNIn(2+2*Outs:2+3*Outs-1)=qd';NNIn(2+3*Outs:2+4*Outs-1)=qdp';NNIn(2+4*Outs:2+5*Outs-1)=qdpp';
XX=V'*NNIn';
NNOut=logsig(XX);
NNOut = [1 NNOut']';
%control torques.
tau=Kv*r+W'*NNOut;
%parameter updates
T_w_dot=F*NNOut*r';
for i = 1:Outs
   xdot(W_start+(i-1)*L:W_start+i*L-1) = T_w_dot(:,i)';
end
T_v_dot=G*NNIn'*((diag(NNOut(2:end))*(eye(L-1)-diag(NNOut(2:end))))*W(2:end,:)*r)';
for i = 1:(L-1)
   xdot(V_start+(i-1)*I:V_start+i*I-1) = T_v_dot(:,i)';
end
% Robot Arm Dynamics
% m1 = weight of first link, m2 = weight of second link, m3 = payload
% weight, a1 = length of first link, a2 = length of second link
m1=0.8; m2=2.3; m3=2.7; a1=1; a2=1; g=9.8; % arm parameters
```

```
if t > 14
m3 = 2.7+1.5;
end
%Inertia M(q)
M11=(m3+m2+0.25*m1)*a1^2*(cos(q(2)))^2 + (m3+0.25*m2)*a2^2*(sin(q(2)+q(3)))^2 ...
    + (m3+m2)*a1*a2*cos(q(2))*sin(q(2)+q(3)) + 1;
M22=(m3+m2+0.25*m1)*a1^2 + (m3+0.25*m2)*a2^2 + (2*m3+m2)*a1*a2*sin(q(3)) + 1;
M23=(m3+0.25*m2)*a2^2 + (m3 + 0.5*m2)*a1*a2*sin(q(3)) + 1;
M32=M23;
M33=(m3+0.25*m2)*a2^2 + 1;
M=[M11 0 0; 0 M22 M23; 0 M32 M33];
% Coriolis/centripetal vector V(q,qdot)
V1=qp(1)*qp(2)*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3)))) - (m3+m2+0.25*m1)*a1^3*sin(2*q(2)) ...
   + (m3+m2)*a1*a2*cos(2*q(2)+q(3)) + qp(1)*qp(3)*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3)))) ...
   + (m3+m2)*a1*a2*cos(q(2))*cos(q(2)+q(3));
V2=qp(2)*qp(3)*(-(2*m3+m2)*a1*a2*cos(q(3))) + q(3)^2*(-(m3+0.5*m2)*a1*a2*cos(q(3))) ...
   + 0.5*qp(1)^2*((m3+m2+0.25*m1)*a1^2*sin(2*q(2)) + (m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3))) ...
   + (m3+0.5*m2)*a1*a2*cos(2*q(2)+q(3)));
V3=-0.5*qp(1)^2*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3))) + (m3 +
m2)*a1*a2*cos(q(2))*cos(q(2)+q(3))) ...
   - (0.5*qp(2)^2+qp(2)*qp(3))*(2*m3+m2)*a1*a2*cos(q(3));
V = [V1 V2 V3]';
% Gravity vector G(q)
G1=0;
G2=g*((m3+m2+0.5*m1)*a1*cos(q(2)) + (m3+0.5*m2)*a2*sin(q(2)+q(3)));
G3=g*(m3+0.5*m2)*a2*sin(q(2)+q(3));
G = [G1 G2 G3]';
% Manipulator Dynamics
qpp = M\(tau - V - G);
%state equations
xdot(1)=x(4);
xdot(2)=x(5);
xdot(3)=x(6);
xdot(4)=qpp(1);
xdot(5)=qpp(2);
xdot(6)=qpp(3);
xdot1=xdot';


function xdot1=robadaptNNAT(t,x)
%-------------------------------------------------------------------------
% NN control with augmented weight tuning
%-------------------------------------------------------------------------
global angles
global angles_t
global I
global Outs
%Compute desired trajectory
angles_idx=round(t/angles_t(2))+1;
qd=angles(angles_idx,1:3)';
if(angles_idx > 1)
   qdp=((angles(angles_idx,1:3)-angles(angles_idx-1,1:3))/angles_t(2))';
   if(angles_idx > 2)
       qdpp=(((angles(angles_idx,1:3)-angles(angles_idx-1,1:3)) -
(angles(angles_idx-1,1:3)-angles(angles_idx-2,1:3)))/angles_t(2)^2)';
   else
       qdpp=zeros(Outs,1);
   end
else
   qdp=zeros(Outs,1);
   qdpp=zeros(Outs,1);
end
```

```
%Neural Net
W_start = Outs*2 +1;
W=[x(W_start:W_start+I-1) x(W_start+I:W_start+2*I-1) x(W_start+2*I:W_start+3*I-1)];
q=[x(1) x(2) x(3)]';
qp=[x(4) x(5) x(6)]';
%Adaptive control input
Kv=20*eye(Outs); lam=5*eye(Outs); F=50*eye(I); kappa=0.1;% controller parameters
%tracking errors
e=qd-q;
ep=qdp-qp;
r=ep+lam*e;
%compute regression matrix
%f=qdpp+lam*ep;
NNIn(1)=1;NNIn(2:2+Outs-1)=e';NNIn(2+Outs:2+2*Outs-1)=ep';
NNIn(2+2*Outs:2+3*Outs-1)=qd';NNIn(2+3*Outs:2+4*Outs-1)=qdp';NNIn(2+4*Outs:2+5*Outs-1)=qdpp';
%XX=V'*NNIn';
NNOut=logsig(NNIn');
NNOut(1) = 1;
%control torques. Parameter estimates are [x(5) x(6)]'
% tau=Kv*r;
tau=Kv*r+W'*NNOut;
%parameter updates
T_w_dot=F*NNOut*r'-kappa*F*norm(r)*W;
for i = 1:Outs
   xdot(W_start+(i-1)*I:W_start+i*I-1) = T_w_dot(:,i)';
end
% Robot Arm Dynamics
% m1 = weight of first link, m2 = weight of second link, m3 = payload
% weight, a1 = length of first link, a2 = length of second link
m1=0.8; m2=2.3; m3=2.7; a1=1; a2=1; g=9.8; % arm parameters
if t > 14
m3 = 2.7+1.5;
end
%Inertia M(q)
M11=(m3+m2+0.25*m1)*a1^2*(cos(q(2)))^2 + (m3+0.25*m2)*a2^2*(sin(q(2)+q(3)))^2 ...
    + (m3+m2)*a1*a2*cos(q(2))*sin(q(2)+q(3)) + 1;
M22=(m3+m2+0.25*m1)*a1^2 + (m3+0.25*m2)*a2^2 + (2*m3+m2)*a1*a2*sin(q(3)) + 1;
M23=(m3+0.25*m2)*a2^2 + (m3 + 0.5*m2)*a1*a2*sin(q(3)) + 1;
M32=M23;
M33=(m3+0.25*m2)*a2^2 + 1;
M=[M11 0 0; 0 M22 M23; 0 M32 M33];
% Coriolis/centripetal vector V(q,qdot)
V1=qp(1)*qp(2)*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3)))) - (m3+m2+0.25*m1)*a1^3*sin(2*q(2)) ...
   + (m3+m2)*a1*a2*cos(2*q(2)+q(3)) + qp(1)*qp(3)*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3)))) ...
   + (m3+m2)*a1*a2*cos(q(2))*cos(q(2)+q(3));
V2=qp(2)*qp(3)*(-(2*m3+m2)*a1*a2*cos(q(3))) + q(3)^2*(-(m3+0.5*m2)*a1*a2*cos(q(3))) ...
   + 0.5*qp(1)^2*((m3+m2+0.25*m1)*a1^2*sin(2*q(2)) + (m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3))) ...
   + (m3+0.5*m2)*a1*a2*cos(2*q(2)+q(3)));
V3=-0.5*qp(1)^2*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3))) + (m3 +
m2)*a1*a2*cos(q(2))*cos(q(2)+q(3))) ...
   - (0.5*qp(2)^2+qp(2)*qp(3))*(2*m3+m2)*a1*a2*cos(q(3));
V = [V1 V2 V3]';
% Gravity vector G(q)
G1=0;
G2=g*((m3+m2+0.5*m1)*a1*cos(q(2)) + (m3+0.5*m2)*a2*sin(q(2)+q(3)));
G3=g*(m3+0.5*m2)*a2*sin(q(2)+q(3));
G = [G1 G2 G3]';
% Manipulator Dynamics
qpp = M\(tau - V - G);
%state equations
xdot(1)=x(4);
xdot(2)=x(5);
```

```
xdot(3)=x(6);
xdot(4)=qpp(1);
xdot(5)=qpp(2);
xdot(6)=qpp(3);
xdot1=xdot';


function xdot1=robadaptNN(t,x)
%-------------------------------------------------------------------------
% NN control with Backprop weight tuning
%-------------------------------------------------------------------------
global angles
global angles_t
global I
global Outs
%Compute desired trajectory
angles_idx=round(t/angles_t(2))+1;
qd=angles(angles_idx,1:3)';
if(angles_idx > 1)
   qdp=((angles(angles_idx,1:3)-angles(angles_idx-1,1:3))/angles_t(2))';
   if(angles_idx > 2)
       qdpp=(((angles(angles_idx,1:3)-angles(angles_idx-1,1:3)) -
(angles(angles_idx-1,1:3)-angles(angles_idx-2,1:3)))/angles_t(2)^2)';
   else
       qdpp=zeros(Outs,1);
   end
else
   qdp=zeros(Outs,1);
   qdpp=zeros(Outs,1);
end
%Neural Net
W_start = Outs*2 +1;
W=[x(W_start:W_start+I-1) x(W_start+I:W_start+2*I-1) x(W_start+2*I:W_start+3*I-1)];
q=[x(1) x(2) x(3)]';
qp=[x(4) x(5) x(6)]';
%Adaptive control input
Kv=20*eye(Outs); lam=5*eye(Outs); F=50*eye(I); % controller parameters
%tracking errors
e=qd-q;
ep=qdp-qp;
r=ep+lam*e;
%compute regression matrix
%f=qdpp+lam*ep;
NNIn(1)=1;NNIn(2:2+Outs-1)=e';NNIn(2+Outs:2+2*Outs-1)=ep';
NNIn(2+2*Outs:2+3*Outs-1)=qd';NNIn(2+3*Outs:2+4*Outs-1)=qdp';NNIn(2+4*Outs:2+5*Outs-1)=qdpp';
%XX=V'*NNIn';
NNOut=logsig(NNIn');
NNOut(1) = 1;
%control torques. Parameter estimates are [x(5) x(6)]'
tau=Kv*r+W'*NNOut;
%parameter updates
T_w_dot=F*NNOut*r';
for i = 1:Outs
   xdot(W_start+(i-1)*I:W_start+i*I-1) = T_w_dot(:,i)';
end
% Robot Arm Dynamics
% m1 = weight of first link, m2 = weight of second link, m3 = payload
% weight, a1 = length of first link, a2 = length of second link
m1=0.8; m2=2.3; m3=2.7; a1=1; a2=1; g=9.8; % arm parameters
if t > 14
m3 = 2.7+1.5;
end
```

```
%Inertia M(q)
M11=(m3+m2+0.25*m1)*a1^2*(cos(q(2)))^2 + (m3+0.25*m2)*a2^2*(sin(q(2)+q(3)))^2 ...
    + (m3+m2)*a1*a2*cos(q(2))*sin(q(2)+q(3)) + 1;
M22=(m3+m2+0.25*m1)*a1^2 + (m3+0.25*m2)*a2^2 + (2*m3+m2)*a1*a2*sin(q(3)) + 1;
M23=(m3+0.25*m2)*a2^2 + (m3 + 0.5*m2)*a1*a2*sin(q(3)) + 1;
M32=M23;
M33=(m3+0.25*m2)*a2^2 + 1;
M=[M11 0 0; 0 M22 M23; 0 M32 M33];
% Coriolis/centripetal vector V(q,qdot)
V1=qp(1)*qp(2)*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3)))) - (m3+m2+0.25*m1)*a1^3*sin(2*q(2)) ...
   + (m3+m2)*a1*a2*cos(2*q(2)+q(3)) + qp(1)*qp(3)*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3)))) ...
   + (m3+m2)*a1*a2*cos(q(2))*cos(q(2)+q(3));
V2=qp(2)*qp(3)*(-(2*m3+m2)*a1*a2*cos(q(3))) + q(3)^2*(-(m3+0.5*m2)*a1*a2*cos(q(3))) ...
   + 0.5*qp(1)^2*((m3+m2+0.25*m1)*a1^2*sin(2*q(2)) + (m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3))) ...
   + (m3+0.5*m2)*a1*a2*cos(2*q(2)+q(3)));
V3=-0.5*qp(1)^2*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3))) + (m3 +
m2)*a1*a2*cos(q(2))*cos(q(2)+q(3))) ...
   - (0.5*qp(2)^2+qp(2)*qp(3))*(2*m3+m2)*a1*a2*cos(q(3));
V = [V1 V2 V3]';
% Gravity vector G(q)
G1=0;
G2=g*((m3+m2+0.5*m1)*a1*cos(q(2)) + (m3+0.5*m2)*a2*sin(q(2)+q(3)));
G3=g*(m3+0.5*m2)*a2*sin(q(2)+q(3));
G = [G1 G2 G3]';
% Manipulator Dynamics
qpp = M\(tau - V - G);
%state equations
xdot(1)=x(4);
xdot(2)=x(5);
xdot(3)=x(6);
xdot(4)=qpp(1);
xdot(5)=qpp(2);
xdot(6)=qpp(3);
xdot1=xdot';


function xdot1=robadaptPD(t,x)
%-------------------------------------------------------------------------
% NN control with Backprop weight tuning
%-------------------------------------------------------------------------
global angles
global angles_t
global Outs
%Compute desired trajectory
angles_idx=round(t/angles_t(2))+1;
qd=angles(angles_idx,1:3)';
if(angles_idx > 1)
   qdp=((angles(angles_idx,1:3)-angles(angles_idx-1,1:3))/angles_t(2))';
   if(angles_idx > 2)
       qdpp=(((angles(angles_idx,1:3)-angles(angles_idx-1,1:3)) -
(angles(angles_idx-1,1:3)-angles(angles_idx-2,1:3)))/angles_t(2)^2)';
   else
       qdpp=zeros(Outs,1);
   end
else
   qdp=zeros(Outs,1);
   qdpp=zeros(Outs,1);
end
q=[x(1) x(2) x(3)]';
qp=[x(4) x(5) x(6)]';
%Adaptive control input
Kv=30*eye(Outs); lam=5*eye(Outs); % controller parameters
```

```
%tracking errors
e=qd-q;
ep=qdp-qp;
r=ep+lam*e;
%control torques. Parameter estimates are [x(5) x(6)]'
tau=Kv*r;
% Robot Arm Dynamics
% m1 = weight of first link, m2 = weight of second link, m3 = payload
% weight, a1 = length of first link, a2 = length of second link
m1=0.8; m2=2.3; m3=2.7; a1=1; a2=1; g=9.8; % arm parameters
if t > 14
m3 = 2.7+1.5;
end
%Inertia M(q)
M11=(m3+m2+0.25*m1)*a1^2*(cos(q(2)))^2 + (m3+0.25*m2)*a2^2*(sin(q(2)+q(3)))^2 ...
    + (m3+m2)*a1*a2*cos(q(2))*sin(q(2)+q(3)) + 1;
M22=(m3+m2+0.25*m1)*a1^2 + (m3+0.25*m2)*a2^2 + (2*m3+m2)*a1*a2*sin(q(3)) + 1;
M23=(m3+0.25*m2)*a2^2 + (m3 + 0.5*m2)*a1*a2*sin(q(3)) + 1;
M32=M23;
M33=(m3+0.25*m2)*a2^2 + 1;
M=[M11 0 0; 0 M22 M23; 0 M32 M33];
% Coriolis/centripetal vector V(q,qdot)
V1=qp(1)*qp(2)*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3)))) - (m3+m2+0.25*m1)*a1^3*sin(2*q(2)) ...
   + (m3+m2)*a1*a2*cos(2*q(2)+q(3)) + qp(1)*qp(3)*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3)))) ...
   + (m3+m2)*a1*a2*cos(q(2))*cos(q(2)+q(3));
V2=qp(2)*qp(3)*(-(2*m3+m2)*a1*a2*cos(q(3))) + q(3)^2*(-(m3+0.5*m2)*a1*a2*cos(q(3))) ...
   + 0.5*qp(1)^2*((m3+m2+0.25*m1)*a1^2*sin(2*q(2)) + (m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3))) ...
   + (m3+0.5*m2)*a1*a2*cos(2*q(2)+q(3)));
V3=-0.5*qp(1)^2*((m3+0.25*m2)*a2^2*sin(2*(q(2)+q(3))) + (m3 +
m2)*a1*a2*cos(q(2))*cos(q(2)+q(3))) ...
   - (0.5*qp(2)^2+qp(2)*qp(3))*(2*m3+m2)*a1*a2*cos(q(3));
V = [V1 V2 V3]';
% Gravity vector G(q)
G1=0;
G2=g*((m3+m2+0.5*m1)*a1*cos(q(2)) + (m3+0.5*m2)*a2*sin(q(2)+q(3)));
G3=g*(m3+0.5*m2)*a2*sin(q(2)+q(3));
G = [G1 G2 G3]';
% Manipulator Dynamics
qpp = M\(tau - V - G);
%state equations
xdot(1)=x(4);
xdot(2)=x(5);
xdot(3)=x(6);
xdot(4)=qpp(1);
xdot(5)=qpp(2);
xdot(6)=qpp(3);
xdot1=xdot';
```