Neural Network Adaptive Backstepping Control for a Quadrotor Drone using Lagrange Dynamics

Nathan A. Lutes

***Abstract*—Quadrotor drones are popular gateway platforms for rotorcraft control system design because of their having simple dynamics while still exhibiting representative problems such as underactuation, strong dynamic coupling, multi-input/multi-output design and unknown nonlinearities. As such, much work has been done on controller design for the quadrotor including nonlinear control. This work replicates a previous study which focused on the design of a neuro-adaptive backstepping controller and avoided the challenges presented in previous literature by implementing the backstepping directly on the Lagrangian dynamics. Furthermore, the problem of the dynamics being bilinear in the virtual control is solved using an inverse kinematics approach. The neuro-adaptive portion of the controller stems from the use of two neural networks in the control design which are used to estimate uncertainties and complicated terms within the control architecture, vastly simplifying and robustifying the resulting controller. The realized set of control laws provides for a control system structure consisting of an inner loop for attitude control followed by an outer loop for position control and without needing knowledge of the complicated aerodynamic forces and moments. This paper describes the Lagrangian dynamics of the quadrotor drone, details the control design process, provides a stability analysis based on Lyapunov's direct method proving that the tracking error and weight estimation errors are uniformly ultimately bounded and validates the results via MATLAB simulation involving a trajectory tracking problem.**

I. Introduction

In recent years, the quadrotor drone has been a popular research platform for the development and testing of modern control algorithms. With simpler dynamics than traditional helicopters, quadrotor drones provide an accessible test-bed for new controllers designed for aircraft applications. The popularity of quadrotor drones in commercial and military applications as well as their usefulness in rescue, surveillance, inspection, mapping etc. also behooves control research for this type of robot. Controllers have been designed for quadrotor drones in the form of linearization and linear control design, linear quadratic regulator (LQR), H-infinity state-space design, model-predictive control, feedback linearization and backstepping [6]. However, designing a controller for a quadrotor is not always straightforward. For instance, previous backstepping controllers designed on the state-space form of quadrotor drone dynamics have lead to complex control laws that often require computation of lie derivatives. Performing backstepping control directly on the Lagrangian dynamics, as was done in [7], [3], [2] and [8] can mitigate these challenges. This paper replicates the procedure and simulation done by [4] which also performs backstepping control directly on the Lagrangian dynamics but uses an inverse kinematics solution to deal with the bilinear in the virtual control problem. This way the desired thrust and torques can be calculated and designed separately.

Another challenge in controller design for a quadrotor drone is precise modelling of the complicated aerodynamic forces and moments that are acting on the drone in flight. To remedy this problem, previous researchers have turned to using neural networks to approximate the uncertainties using adaptive control techniques [1], [5]. Neural networks are a natural choice for adaptive control due to their 'universal function approximation' property. That is, their ability to estimate most any function to a degree arbitrary to the parameters of the network. Neural networks in control applications are also tuned 'on-line' or rather their parameters are tuned entirely during the control application without the need for any offline training data. The controller discussed in this paper takes advantage of the powerful nature of neural networks by featuring one in the input thrust controller and another in the torque controller. The neural networks allow for estimation of the most complicated parts of the controller structure which vastly simplifies the control laws. The stability of the network weights, given the online weight tuning laws, is proven using Lyapunov analysis.

The rest of the paper proceeds as follows: in section 2, the quadrotor dynamics are discussed. In section 3, the design objective and controller design, including the neural network approximations, are presented. In section 4, the weight tuning laws are presented and the stability of the entire system, including the neural network weights, is analyzed using Lyapunov's direct method. In section 5, a MATLAB simulation of the controller and the results are presented and discussed and the paper concludes in section 6.

II. Quadrotor Dynamics

In this section, the Lagrangian dynamics of the quadrotor drone will be presented as they are described in [4]. Because of the quadrotor design, the dynamics of the system are not complicated. The different attitude orientations of the craft are obtained by varying the individual motor speeds thus creating torques on the different rotational axes. The thrust force combined with the attitude vector creates the translational motion. Therefore, the control objective will be split into finding the total thrust required followed by the total torque needed to produce the required attitude which leads to the desired translational movement. Since the rotor dynamics are relatively much faster than the system dynamics, they are neglected.

The states of the quadrotor can be described as $q = (x, y, z, \psi, \theta, \phi)$ as well as their time derivatives $\dot{q} = (\dot{x}, \dot{y}, \dot{z}, \dot{\psi}, \dot{\theta}, \dot{\phi})$. The coordinates (x, y, z) represent the quadrotor's relative position in the euclidean space and (ψ, θ, ϕ) represent the orientation of the craft namely yaw, pitch and roll. Defining $\zeta = (x, y, z)^T$ and $\eta = (\psi, \theta, \phi)^T$, the dynamic model of the quadrotor can be written in terms of the following subsystems:

$$m\ddot{\zeta} + \begin{bmatrix} 0 \\ 0 \\ mg \end{bmatrix} = F_d + A_{xyz} \tag{1}$$

$$J(\eta)\ddot{\eta} + C(\eta, \dot{\eta})\dot{\eta} = \tau + A_\eta \tag{2}$$

where

$$C(\eta, \dot{\eta}) = \frac{d}{dt}\{J(\eta)\}\dot{\eta} - \frac{1}{2}\frac{\partial}{\partial \eta}\left(\dot{\eta}^T J(\eta)\dot{\eta}\right)$$

is the coriolis/centripetal term and A_{xyz} and A_η are unmodelled, state-dependent disturbances. The effective inertia matrix ($J(\eta)$) is defined as:

$$J(\eta) = T_\eta^T \Sigma T_\eta = J \tag{3}$$

where

$$T_\eta = \begin{pmatrix} -sin(\theta) & 0 & 1 \\ cos(\theta)sin(\psi) & cos(\psi) & 0 \\ cos(\theta)cos(\psi) & -sin(\psi) & 0 \end{pmatrix}$$

is a rotation transformation and Σ is the quadrotor inertia matrix. It is important to note that while Σ is constant, that J is not. Using the inertia matrix, Σ, $C(\eta, \dot{\eta})$ can also be defined as:

$$C(\eta, \dot{\eta}) = 2T_\eta^T \Sigma \dot{T}_\eta - \frac{1}{2}\frac{\partial}{\partial \eta}\left(\dot{\eta}^T J(\eta)\dot{\eta}\right) \tag{4}$$

Using this specific representation (which is not unique), it can be proven that $\frac{d}{dt}J(\eta) - 2C(\eta, \dot{\eta})$ is skew-symmetric. This will be an important property in the stability proofs following in a later section.

As seen in (1), F_d is a force vector acting as the virtual control into the position subsystem. F_d is related to the actual control via:

$$F_d = u \begin{pmatrix} -sin(\theta) \\ cos(\theta)sin(\phi) \\ cos(\theta)cos(\phi) \end{pmatrix} \tag{5}$$

The total thrust u combined with the torque τ creates the control inputs $(u, \tau^T)^T$ for the system.

III. Neuro - Adaptive Backstepping Controller Design

The design strategy for controlling the quadcopter for a position tracking problem is to use backstepping control directly on the Lagrangian dynamics. First, the position subsystem is used to determine the required virtual control signal F_d. Then, the required thrust u is derived from F_d as well as the required orientation of the copter (ψ, θ, ϕ). Afterwards, the τ required to fulfill this orientation can be calculated using the rotational subsystem.

First, the position tracking error is defined as:

$$e_1 = \begin{bmatrix} x_d - x \\ y_d - y \\ z_d - z \end{bmatrix}^T = \zeta_d - \zeta \tag{6}$$

which allows us to define the position sliding mode error signal:

$$r_1 = \dot{e}_1 + \Lambda_1 e_1 \tag{7}$$

where Λ_1 is a diagonal, positive definite design parameter matrix making (7) a stable system guaranteeing the bound on e_1 if r_1 is bounded.

Using (7), the position error dynamics become:

$$m\dot{r}_1 = m\ddot{e}_1 + m\Lambda_1 \dot{e}_1 \tag{8}$$

which becomes

$$m\dot{r}_1 = m \begin{bmatrix} \ddot{x}_d \\ \ddot{y}_d \\ \ddot{z}_d \end{bmatrix} - m \begin{bmatrix} \ddot{x} \\ \ddot{y} \\ \ddot{z} \end{bmatrix} + m\Lambda_1(r_1 - \Lambda_1 e_1)$$

$$m\dot{r}_1 = m\ddot{\zeta}_d - F_d - m_g - A_{xyz} + m\Lambda_1 r_1 - m\Lambda_1^2 e_1 \tag{9}$$

where $m_g = \begin{pmatrix} 0 \\ 0 \\ -mg \end{pmatrix}$. Then the ideal virtual force which stabilizes (9) is:

$$\hat{F}_d = m\ddot{\zeta}_d - m\Lambda_1^2 e_1 - m_g - \hat{A}_{xyz} + K_{r_1} r_1 + K_{i_1}\int_0^t r_1 dt \tag{10}$$

where $\hat{A}_{xyz}$ is an approximation of A_{xyz} and K_{r_1} and K_{i_1} are diagonal, positive definite control gain matrices. The approximation is done using a neural network which is a machine learning algorithm well known for its universal function approximation properties. The neural network estimation can be described mathematically as:

$$\nu_{NN_1} \triangleq -A_{xyz} = W_1^T \mu_1(X_1) + \epsilon_1 \tag{11}$$

where W_1 is an unknown matrix of ideal weights that provide the best possible function approximation, $X_1 = \left(\zeta, r_1, \eta, \dot{\eta}\right)^T$ is the input and ϵ_1 is the ideal estimation error which is assumed bounded by $\|\epsilon_1\| \leq \epsilon_{N_1}$. However, it cannot be readily assumed that the actual neural network weights are equivalent to the ideal weights, so the actual estimation is described as:

$$\hat{\nu}_{NN_1} = -\hat{A}_{xyz} = \hat{W}_1^T \mu_1(X_1) \tag{12}$$

The weight tuning equation will be described in the section detailing the stability analysis. Now the desired virtual input can be described as:

$$\hat{F}_d = m\ddot{\zeta}_d - m\Lambda_1^2 e_1 - m_g + \hat{W}_1^T \mu_1(X_1) + K_{r_1} r_1 + K_{i_1}\int_0^t r_1 dt \tag{13}$$

Substituting (13) into (9), the error dynamics become:

$$m\dot{r}_1 = -(K_{r_1} - m\Lambda_1)r_1 - \tilde{A}_{xyz} - K_{i_1}\int_0^t r_1 dt \tag{14}$$

$$\tilde{A}_{xyz} = A_{xyz} - \hat{A}_{xyz}$$
$$= W_1^T \mu_1(X_1) - \hat{W}_1^T \mu(X_1) + \epsilon_1 = \tilde{W}_1^T \mu_1 + \epsilon_1$$
$$m\dot{r}_1 = -(K_{r_1} - m\Lambda_1)r_1 + \tilde{W}_1^T \mu_1(X_1) - K_{i_1} \int_0^t r_1 dt \quad (15)$$

Now that the desired virtual input for the position system has been defined, the second stage of the backstepping design can commence: finding the desired orientation and designing τ to achieve it. To find the desired orientation, $(\psi_d, \theta_d, \phi_d)$ must be computed from (5). This is done in [4] by using robot inverse kinematics, detailed below. Define:

$$\hat{F}_d = \begin{pmatrix} -u_d sin(\theta_d) \\ u_d cos(\theta_d) sin(\phi_d) \\ u_d cos(\theta_d) cos(\phi_d) \end{pmatrix} \quad (16)$$
$$a = u_d sin(\theta_d), \qquad b = u_d cos(\theta_d)$$

Then
$$\begin{pmatrix} -a \\ b sin(\phi_d) \\ b cos(\phi_d) \end{pmatrix} = \hat{F}_d(t) = \begin{pmatrix} f_{x_d}(t) \\ f_{y_d}(t) \\ f_{z_d}(t) \end{pmatrix} \quad (17)$$
$$a = -f_{x_d}$$
$$b^2 sin^2(\psi_d) + b^2 cos^2(\psi_d) = f_{y_d}^2(t) + f_{z_d}^2(t)$$
$$b = \sqrt{f_{y_d}^2(t) + f_{z_d}^2(t)}$$

From the definition of a and b:
$$a^2 + b^2 = (u_d sin(\theta_d))^2 + (u_d cos(\theta_d))^2 \quad (18)$$
$$u_d^2 = a^2 + b^2$$
$$u_d = \sqrt{a^2 + b^2} \quad (19)$$

and
$$\frac{a}{b} = \frac{u_d sin(\theta_d)}{u_d cos(\theta_d)} = tan(\theta_d) \quad (20)$$
$$\theta_d = tan^{-1}\Big(\frac{a}{b}\Big) \quad (21)$$

Finally,
$$\frac{f_{y_d}}{f_{z_d}} = \frac{b sin(\phi_d)}{b cos(\phi_d)} = tan(\phi_d) \quad (22)$$
$$\phi_d = tan^{-1}\Big(\frac{f_{y_d}}{f_{z_d}}\Big) \quad (23)$$

and thus (u_d, θ_d, ϕ_d) can all be extracted from the virtual input. Note that ψ_d is absent because it is not present in the expression for $\hat{F}_d$. Therefore, ψ_d can be given as a desired reference input. With the desired orientation defined, the appropriate τ to realize this orientation can be derived.

First, the orientation tracking error is defined as:
$$e_2 = (\psi_d - \psi, \theta_d - \theta, \phi_d - \phi)^T \quad (24)$$

with the sliding mode error subsequently defined as:
$$r_2 = \dot{e}_2 + \Lambda_2 e_2 \quad (25)$$

where Λ_2 is a diagonal, positive definite design matrix similar to Λ_1. As was the case with r_1, when r_2 goes to zero, the error will become a stable system with first order, asymptotic convergence characteristics. Note that the derivative of the tracking error $\dot{e}_2$ has not been rigorously defined however in practice this quantity can be approximated using a backward difference method.

Using (24) and (25), the error dynamics for the rotational system can be described as:
$$J\dot{r}_2 = J\ddot{e}_2 + J\Lambda_2 \dot{e}_2$$
$$J\dot{r}_2 = J\ddot{\eta}_d - \Big\{\tau - C(\eta, \dot{\eta})\dot{\eta} + A_\eta\Big\} + J\Lambda_2(r_2 - \Lambda_2 e_2) \quad (26)$$

and from (25)
$$\dot{\eta} = -r_2 + \dot{\eta}_d + \Lambda_2 e_2 \quad (27)$$

so then substituting (27) into (26)
$$J\dot{r}_2 = J\ddot{\eta}_d - \Big\{\tau - C(\eta, \dot{\eta})\Big(- r_2 + \dot{\eta}_d + \Lambda_2 e_2\Big) + A_\eta\Big\} + J\Lambda_2(r_2 - \Lambda_2 e_2)$$

$$J\dot{r}_2 = -\tau - C(\eta, \dot{\eta})r_2 + \Big\{J\ddot{\eta}_d - A_\eta + C(\eta, \dot{\eta})\Big(\dot{\eta}_d + \Lambda_2 e_2\Big) - J\Lambda_2^2 e_2\Big\} + J\Lambda_2 r_2 \quad (28)$$

Now introduce a second neural network to approximate $\Big\{J\ddot{\eta}_d - A_\eta + C(\eta, \dot{\eta})\Big(\dot{\eta}_d + \Lambda_2 e_2\Big) - J\Lambda_2^2 e_2\Big\}$ such that:
$$\nu_{NN_2} \triangleq \Big\{J\ddot{\eta}_d - A_\eta + C(\eta, \dot{\eta})\Big(\dot{\eta}_d + \Lambda_2 e_2\Big) - J\Lambda_2^2 e_2\Big\} = W_2^T \mu_2(X_2) + \epsilon_2 \quad (29)$$

Similarly to W_1, W_2 is the unknown ideal weight matrix and ϵ_2 is the subsequent bounded approximation error term ($\|\epsilon_2\| \leq \epsilon_{N_2}$). X_2 is the input to the second neural network defined as $X_2 = (\eta, r_1, r_2)^T$. Since the true weights cannot be assumed to be equal to the ideal weights, the approximation is represented mathematically as:
$$\hat{\nu}_{NN_2} = \hat{W}_2^T \mu_2(X_2) \quad (30)$$

Now the τ that stabilizes the error dynamics can be defined as:
$$\tau = \hat{\nu}_{NN_2} + K_{r_2} r_2 + K_{i_2} \int_0^t r_2 dt \quad (31)$$

where K_{r_2} and K_{i_2} are diagonal, positive definite gain matrices. Then, the error dynamics reduce to:
$$J\dot{r}_2 = \tilde{\nu}_{NN_2} + J\Lambda_2 r_2 - C(\eta, \dot{\eta})r_2 - K_{r_2} r_2 - K_{i_2} \int_0^t r_2 dt \quad (32)$$

where
$$\tilde{\nu}_{NN_2} = \nu_{NN_2} - \hat{\nu}_{NN_2}$$
$$= W_2^T \mu_2(X_2) - \hat{W}_2^T \mu_2(X_2) + \epsilon_2 = \tilde{W}_2^T \mu_2(X_2) + \epsilon_2 \quad (33)$$

which allows the error dynamics to be expressed as:

$$J\dot{r}_2 = \tilde{W}_2^T \mu_2(X_2) - \Big\{K_{r_2} + C(\eta,\dot{\eta}) - J\Lambda_2\Big\} r_2 - K_{i_2}\int_0^t r_2 dt + \epsilon_2 \quad (34)$$

With the controllers and error systems being defined, now the stability analysis can be conducted leading to the stable update laws for the neural networks.

IV. Stability Analysis and NN Tuning

Lyapunov analysis is used in this section to verify the stability of the error dynamics and thus prove the ability of the controller. The weight update laws for the neural networks are also provided in this section and used in the stability analysis. The main result from this section is summarized in the following theorem.

Theorem 1. *Given the system described in (1) and (2) and using the control laws defined in (19) and (31), under the following assumptions:*

$$K_{r_1} - m\Lambda_1 > 0, \qquad K_{r_2} - J\Lambda_2 > 0$$

and using the neural network weight tuning laws:

$$\dot{\hat{W}}_1 = F_1 \mu_1 r_1^T - \kappa \|r\| F_1 \hat{W}_1, \quad (35)$$

$$\dot{\hat{W}}_2 = F_2 \mu_2 r_2^T - \kappa \|r\| F_2 \hat{W}_2 \quad (36)$$

(where F_1, F_2 are positive definite, symmetric matrices) then the errors r_1, r_2 and the NN weight estimation errors $\tilde{W}_1, \tilde{W}_2$ are UUB with bounds given. Furthermore, r_1, r_2 can be made arbitrarily small with suitable choices of the design parameters $K_{r_1}, K_{r_2}, \Lambda_1, \Lambda_2$

Proof. Define

$$r = \begin{bmatrix} r_1 \\ r_2 \end{bmatrix}, \quad \hat{W} = \begin{pmatrix} \hat{W}_1 \\ \hat{W}_2 \end{pmatrix}$$

$$F = F^T = \begin{bmatrix} F_1 & 0 \\ 0 & F_2 \end{bmatrix} > 0$$

$$\mu_r = \begin{pmatrix} \mu_1 r_1^T \\ \mu_2 r_2^T \end{pmatrix}, \quad \kappa > 0$$

Consider the Lyapunov function candidate:

$$L = \frac{1}{2} r_1^T m r_1 + \frac{1}{2} r_2^T J r_2 + \frac{1}{2}\left[\int_0^t r_1^T dt\right] K_{i_1} \left[\int_0^t r_1 dt\right] + \frac{1}{2}\left[\int_0^t r_2^T dt\right] K_{i_2} \left[\int_0^t r_2 dt\right] + \frac{1}{2} tr\Big\{\tilde{W}^T F^{-1} \tilde{W}\Big\} \quad (37)$$

Taking the derivative

$$\dot{L} = -r^T \begin{bmatrix} K_{r_1} - m\Lambda_1 & 0 \\ 0 & K_{r_2} - J\Lambda_2 \end{bmatrix} r + r^T \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \end{bmatrix} \frac{1}{2} r_2^T \dot{J} r_2 - r_2^T C r_2 + tr\Big\{\tilde{W}^T\Big(F^{-1}\dot{\tilde{W}} + \mu_r\Big)\Big\} \quad (38)$$

Using

$$\dot{\tilde{W}} = -F\mu_r + \kappa \|r\| F \hat{W} \quad (39)$$

Substituting (39) into (38):

$$\dot{L} \le -r^T K_\nu r + r^T \epsilon_\nu + \frac{1}{2} r^T (\dot{J} - 2C) r + \kappa \|r\| tr\Big\{\tilde{W}^T \hat{W}\Big\} \quad (40)$$

where

$$K_\nu = \begin{bmatrix} K_{r_1} - m\Lambda_1 & 0 \\ 0 & K_{r_2} - J\Lambda_2 \end{bmatrix}, \qquad \epsilon_\nu = \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \end{bmatrix} \quad (41)$$

with $K_\nu > 0$ by assumption. Furthermore:

$$\dot{L} \le -r^T K_\nu r + r^T \epsilon_\nu + \frac{1}{2} r^T (\dot{J} - 2C) r + \kappa \|r\| tr\Big\{\tilde{W}^T\Big(W - \tilde{W}\Big)\Big\} \quad (42)$$

where $(\dot{J} - 2C)$ is a skew symmetric matrix and therefore $r^T(\dot{J} - 2C)r = 0$. Using the property:

$$tr\big\{\tilde{W}^T(W - \tilde{W})\big\} \le \|\tilde{W}\|_F \|W\|_F - \|\tilde{W}\|_F^2$$

with $\|\cdot\|_F$ being the Frobenius norm, (42) becomes:

$$\dot{L} \le -\|K_\nu\|_{min}\|r\|^2 + \kappa\|r\|\big\{\|\tilde{W}\|_F\|W\|_F - \|\tilde{W}\|_F^2\big\} + \|\epsilon_\nu\|_{max}\|r\|$$

$$\dot{L} \le -\|r\|\Big\{\|K_\nu\|_{min}\|r\| - \kappa\big\{\|\tilde{W}\|_F\|W\|_F - \|\tilde{W}\|_F^2\big\} - \|\epsilon_\nu\|_{max}\Big\} \quad (43)$$

where $\|K_\nu\|_{min}$ is the minimum possible norm of K_ν and $\|\epsilon_\nu\|_{max}$ is the maximum possible norm of ϵ_ν. Finally, (43) can be rewritten as:

$$\dot{L} \le -\|r\|\Big\{\kappa\Big[\|\tilde{W}\|_F - \frac{\|W\|_F}{2}\Big]^2 - \kappa\frac{\|W\|_F^2}{4} + \|K_\nu\|_{min}\|r\| - \|\epsilon_\nu\|_{max}\Big\} \quad (44)$$

If $\|W\|_F \le W_B$ then $\dot{L} \le 0$ iff:

$$\|r\| > \frac{\frac{W_B^2}{4} + \|\epsilon_\nu\|_{max}}{\|K_\nu\|_{min}} \quad (45)$$

$$\|\tilde{W}\|_F > \frac{W_B}{2} + \sqrt{\Big(\frac{W_B^2}{4} + \frac{\|\epsilon_\nu\|_{max}}{\kappa}} \quad (46)$$

Thus $\dot{L}$ is negative outside a compact set and r and $\tilde{W}$ are UUB with the bounds given in (45) and (46). □

V. Simulation Results

To test the efficacy of the controller, a simulation was conducted in MATLAB using the quadrotor model previously discussed. The inertia, mass and gravity parameters used were:

$$\Sigma = diag(5, 5, 5), \quad m = 1, \quad g = 9.81$$

The uncertainty terms used were:

$$A_{xyz} = \begin{pmatrix} \dot{x}^2\dot{\phi} \\ \dot{y}^2\dot{\theta} \\ \dot{z}^2\dot{\psi} \end{pmatrix}, \quad A_{\eta} = \begin{pmatrix} \dot{x}\dot{\phi}^2 \\ \dot{y}\dot{\theta}^2 \\ \dot{z}\dot{\psi}^2 \end{pmatrix}$$

The neural networks had 10 hidden layer neurons and used logistic sigmoid hidden layer activation functions. The tuning parameters used were:

$$F_1 = F_2 = diag(.02) \in \mathcal{R}^{10x10}, \quad \kappa = 0.01$$

The controller gain matrices and sliding mode parameters were:

$$\Lambda_1 = \Lambda_2 = diag(1) \in \mathcal{R}^{3x3}$$
$$K_{r_1} = K_{r_2} = diag(15) \in \mathcal{R}^{3x3}$$
$$K_{i_1} = K_{i_2} = diag(5) \in \mathcal{R}^{3x3}$$

To create the reference trajectories, the method used in [4] was followed. A specific manner of generating a reference trajectory was chosen such that the jerk (acceleration rate of change) would be minimized over the time horizon. This is so the velocities and accelerations would be zero at the end of the trajectory. The trajectory equation chosen was:

$$\dot{x}_d = a_{1,x} + 2a_{2,x}t + 3a_{3,x}t^2 + 4a_{4,x}t^3 + 5a_{5,x}t^4$$
$$\ddot{x}_d = 2a_{2,x} + 6a_{3,x}t + 12a_{4,x}t^2 + 20a_{5,x}t^3$$

If the initial and final velocities and accelerations are zero, one obtains the relation:

$$\begin{bmatrix} d_x \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & t_f & t_f^2 \\ 3 & 4t_f & 5t_f^2 \\ 6 & 12t_f & 20t_f^2 \end{bmatrix}$$

where $d_x = (x_{d_f} - x_{d_0})/t_f^3$. Solving for the coefficients:

$$\begin{bmatrix} a_{3,x} \\ a_{4,x} \\ a_{5,x} \end{bmatrix} = \begin{bmatrix} 1 & t_f & t_f^2 \\ 3 & 4t_f & 5t_f^2 \\ 6 & 12t_f & 20t_f^2 \end{bmatrix}^{-1} \begin{bmatrix} d_x \\ 0 \\ 0 \end{bmatrix}$$

and thus the desired trajectory is given by

$$x_d(t) = x_0 + a_{3,x}t^3 + a_{4,x}t^4 + a_{5,x}t^5$$

Similarly, the desired trajectories for $y_d(t)$ and $z_d(t)$ can be found in much the same way. For the desired trajectory in the simulation, $x_{d,0} = y_{d,0} = z_{d,0} = t_0 = 0$, $t_f = 60$, $x_{d,f} = 20$, $y_{d,f} = 5$, $z_{d,f} = 10$. $\psi_d = 0$ for all t. The initial conditions in the simulation were $\zeta_0 = \eta_0 = 0$.

Figures (1) - (6) display the results of the simulation. Figure (1) displays the desired and actual trajectories and figure (2) displays the x, y and z trajectory tracking errors. From these two figures, it can be seen that the trajectory tracking is excellent with very small error. Figure (3) shows the input thrust which is the total thrust in the z direction. The thrust stays nearly constant with small, sinusoidal variations. Figures (4) - (6) show the applied torque values for each axis. Some minor chattering can be seen in these values with the overall torques being quite small in magnitude.

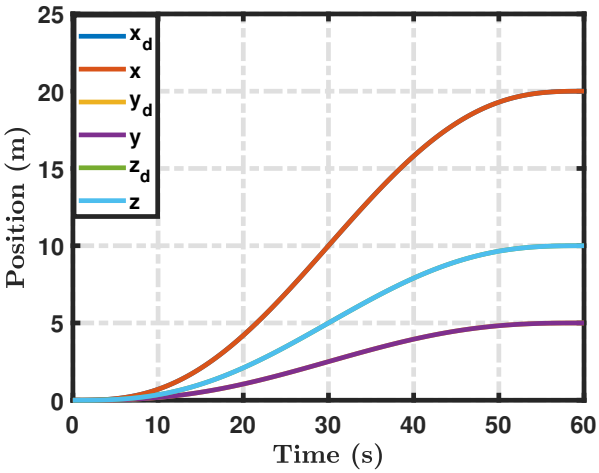

Fig. 1: Desired vs. Actual Trajectories

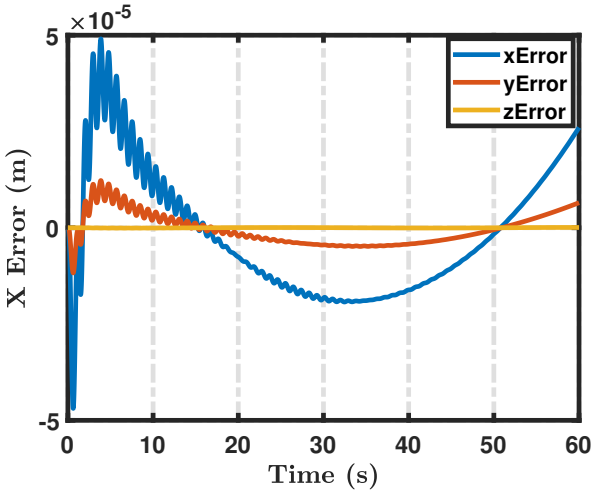

Fig. 2: Trajectory Tracking Errors

VI. Conclusion

This paper detailed a neuro-adaptive backstepping control design that was applied directly to the Lagrangian dynamics of a quadrotor drone. The dynamics of the drone were described, the controller formulation shown and the stability of the controller was proven using Lyapunov analysis. The controller was tested in simulation which produced excellent tracking performance indicating the ability of the neural networks within the controller to quickly learn and account for the unknown dynamics in the system. This approach differs from others in the literature in that the dynamics were in

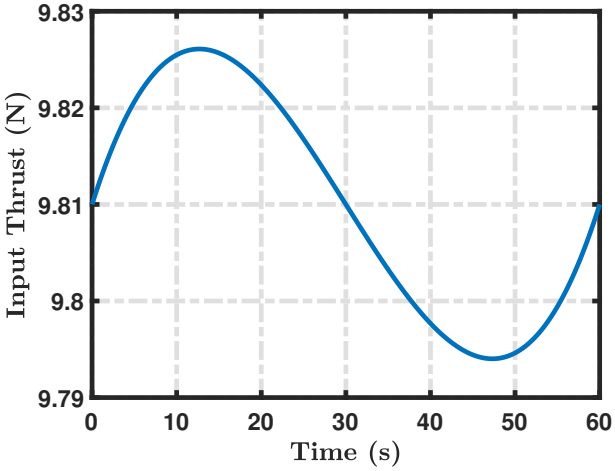

Fig. 3: Input Thrust (u)

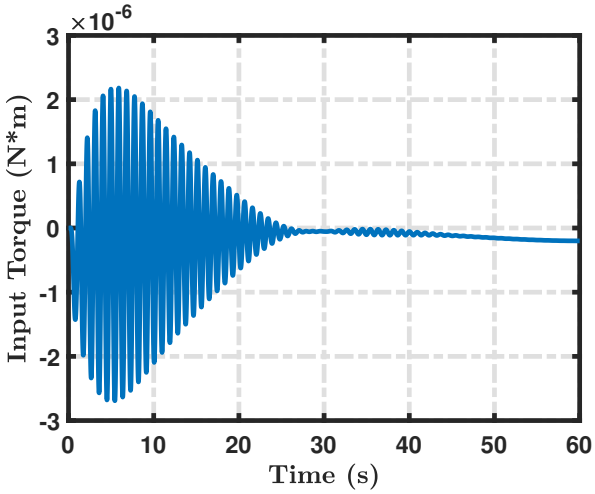

Fig. 4: Z axis Torque

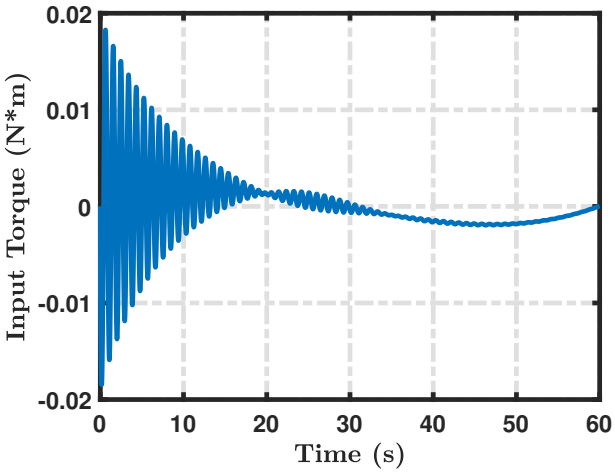

Fig. 5: Y axis Torque

the Lagrangian form and that the bilinear in virtual control

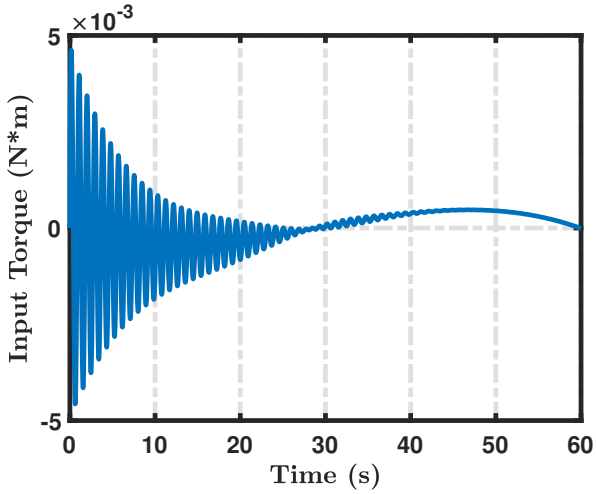

Fig. 6: X axis Torque

problem was solved using inverse robot kinematics.

References

[1] A.j. Calise et al. "Helicopter adaptive flight control using neural networks". In: *Proceedings of 1994 33rd IEEE Conference on Decision and Control* (). DOI: 10.1109/cdc.1994.411659.

[2] P. Castillo, A. Dzul, and R. Lozano. "Real-Time Stabilization and Tracking of a Four-Rotor Mini Rotorcraft". In: *IEEE Transactions on Control Systems Technology* 12.4 (2004), pp. 510–516. DOI: 10.1109/tcst.2004.825052.

[3] Pedro Castillo, Rogelio Lozano, and Dzul Lopez Alejandro Enrique. *Modelling and control of mini-flying machines*. Springer, 2005.

[4] Abhijit Das, Frank Lewis, and Kamesh Subbarao. "Backstepping Approach for Controlling a Quadrotor Using Lagrange Form Dynamics". In: *Journal of Intelligent and Robotic Systems* 56.1-2 (2009), pp. 127–151. DOI: 10.1007/s10846-009-9331-0.

[5] Jay Farrell, Manu Sharma, and Marios Polycarpou. "Backstepping-Based Flight Control with Adaptive Function Approximation". In: *Journal of Guidance, Control, and Dynamics* 28.6 (2005), pp. 1089–1102. DOI: 10.2514/1.13030.

[6] I. Kanellakopoulos, P.v. Kokotovic, and A.s. Morse. "Systematic design of adaptive controllers for feedback linearizable systems". In: *IEEE Transactions on Automatic Control* 36.11 (1991), pp. 1241–1253. DOI: 10.1109/9.100933.

[7] R. Mahony, T. Hamel, and A. Dzul. "Hover control via Lyapunov control for an autonomous model helicopter". In: *Proceedings of the 38th IEEE Conference on Decision and Control (Cat. No.99CH36304)* (). DOI: 10.1109/cdc.1999.827874.

[8] "Stabilization of a mini rotorcraft with four rotors". In: *IEEE Control Systems* 25.6 (2005), pp. 45–55. DOI: 10.1109/mcs.2005.1550152.

```
>> type Project_Main

%Nathan Lutes
%Nonlinear Control Main Script
clear; clc; close all
addpath('S:\Documents\MATLAB\ODE_Solvers')
global e2old tOld e2dotold
%constants and paramters
in1 = 12; in2 = 9; out1 = 3; out2 = 3; L = 10;
e2old = 0; tOld = 0; e2dotold = 0;

%define weights
V1 = 2*rand(in1*L,1)-1; V2 = 2*rand(in2*L,1)-1;
W1 = zeros(L*out1,1); W2 = zeros(L*out2,1);

%simulation
zeta0 = [0,0,0,0,0,0]'; eta0 = [0,0,0,0,0,0]'; intr1 = zeros(3,1); intr2 = zeros(3,1);
x0 = [zeta0; eta0; intr1; intr2; V1; V2; W1; W2];
t1 = 0:0.01:60;
x1 = ode4(@QuadcopterDyn,t1,x0);

%desired trajectory
xdhist = zeros(1,length(t1)); ydhist = zeros(1,length(t1)); zdhist = zeros(1,length(t1));
xd0 = 0; xdf = 20; tf = 60; dx = (xdf - xd0)/(tf^3);
yd0 = 0; ydf = 5; dy = (ydf - yd0)/(tf^3);
zd0 = 0; zdf = 10; dz = (zdf - zd0)/(tf^3);
DTM = [1 tf tf^2; 3 4*tf 5*tf^2; 6 12*tf 20*tf^2];
ax = DTM\[dx 0 0]'; ay = DTM\[dy 0 0]'; az = DTM\[dz 0 0]';
for i = 1:length(t1)
    xdhist(i) = ax'*[t1(i)^3 t1(i)^4 t1(i)^5]';
    ydhist(i) = ay'*[t1(i)^3 t1(i)^4 t1(i)^5]';
    zdhist(i) = az'*[t1(i)^3 t1(i)^4 t1(i)^5]';
end

%plots
Fsize = 15;
Pic_Width=7;
Pic_Height=7;

figure
plot(t1,xdhist,'LineWidth',3)
hold on
plot(t1,x1(:,1),'LineWidth',3)
plot(t1,ydhist,'LineWidth',3)
plot(t1,x1(:,2),'LineWidth',3)
plot(t1,zdhist,'LineWidth',3)
plot(t1,x1(:,3),'LineWidth',3)
hold off
legend('x_d','x','y_d','y','z_d','z')
xlabel('\bf{Time (s)}','FontWeight','bold','FontSize',20,'interpreter','latex')
ylabel('\bf{Position (m)}','FontWeight','bold','FontSize',20,'interpreter','latex')
```

```
ax = gca;ax.LineWidth = 3; ax.FontSize =Fsize; box on; ax.FontWeight='bold';
set(gcf,'PaperUnits','inches','PaperPosition',[0 0 Pic_Width Pic_Height]);
grid;set(gca,'MinorGridLineStyle','-');set(gca,'GridLineStyle','-.');box on;

%error
ex = x1(:,1)-xdhist'; ey = x1(:,2)-ydhist'; ez = x1(:,3)-zdhist';

figure
plot(t1,ex,'LineWidth',3)
hold on
plot(t1,ey,'LineWidth',3)
plot(t1,ez,'LineWidth',3)
hold off
legend('xError', 'yError', 'zError')
xlabel('\bf{Time (s)}','FontWeight','bold','FontSize',20,'interpreter','latex')
ylabel('\bf{X Error (m)}','FontWeight','bold','FontSize',20,'interpreter','latex')
ax = gca;ax.LineWidth = 3; ax.FontSize =Fsize; box on; ax.FontWeight='bold';
set(gcf,'PaperUnits','inches','PaperPosition',[0 0 Pic_Width Pic_Height]);
grid;set(gca,'MinorGridLineStyle','-');set(gca,'GridLineStyle','-.');box on;

%recalculate control
e2old = 0; tOld = 0; e2dotold = 0;
udhist = zeros(1,length(t1));
tauhist = zeros(3,length(t1));
for i = 1:length(t1)
    [ud,tau]=calculateControl(t1(i),x1(i,:)');
    udhist(i) = ud;
    tauhist(:,i) = tau;
end

figure
plot(t1,udhist,'LineWidth',3)
xlabel('\bf{Time (s)}','FontWeight','bold','FontSize',20,'interpreter','latex')
ylabel('\bf{Input Thrust (N)}','FontWeight','bold','FontSize',20,'interpreter','latex')
ax = gca;ax.LineWidth = 3; ax.FontSize =Fsize; box on; ax.FontWeight='bold';
set(gcf,'PaperUnits','inches','PaperPosition',[0 0 Pic_Width Pic_Height]);
grid;set(gca,'MinorGridLineStyle','-');set(gca,'GridLineStyle','-.');box on;

figure
plot(t1,tauhist(1,:),'LineWidth',3)
xlabel('\bf{Time (s)}','FontWeight','bold','FontSize',20,'interpreter','latex')
ylabel('\bf{Input Torque (N*m)}','FontWeight','bold','FontSize',20,'interpreter','latex')
ax = gca;ax.LineWidth = 3; ax.FontSize =Fsize; box on; ax.FontWeight='bold';
set(gcf,'PaperUnits','inches','PaperPosition',[0 0 Pic_Width Pic_Height]);
grid;set(gca,'MinorGridLineStyle','-');set(gca,'GridLineStyle','-.');box on;

figure
plot(t1,tauhist(2,:),'LineWidth',3)
xlabel('\bf{Time (s)}','FontWeight','bold','FontSize',20,'interpreter','latex')
ylabel('\bf{Input Torque (N*m)}','FontWeight','bold','FontSize',20,'interpreter','latex')
ax = gca;ax.LineWidth = 3; ax.FontSize =Fsize; box on; ax.FontWeight='bold';
```

```
set(gcf,'PaperUnits','inches','PaperPosition',[0 0 Pic_Width Pic_Height]);
grid;set(gca,'MinorGridLineStyle','-');set(gca,'GridLineStyle','-.');box on;

figure
plot(t1,tauhist(3,:),'LineWidth',3)
xlabel('\bf{Time (s)}','FontWeight','bold','FontSize',20,'interpreter','latex')
ylabel('\bf{Input Torque (N*m)}','FontWeight','bold','FontSize',20,'interpreter','latex')
ax = gca;ax.LineWidth = 3; ax.FontSize =Fsize; box on; ax.FontWeight='bold';
set(gcf,'PaperUnits','inches','PaperPosition',[0 0 Pic_Width Pic_Height]);
grid;set(gca,'MinorGridLineStyle','-');set(gca,'GridLineStyle','-.');box on;
>> type QuadcopterDyn

function [xdot] = QuadcopterDyn(t,x)
%Quadcopter dynamics for nonlinear control final
global e2old tOld e2dotold
%constants
g = 9.81; L = 10; in1 = 12; in2 = 9; out1 = 3; out2 = 3; m = 1;
Lamb1 = diag(1*ones(1,3)); Lamb2 = diag(1*ones(1,3)); Kr1 = diag(15*ones(1,3));
Kr2 = diag(15*ones(1,3)); Ki1 = diag(5*ones(1,3)); Ki2 = diag(5*ones(1,3));
V1index = 19+L*in1-1; V2index = V1index + L*in2;
W1index = V2index + L*out1;
W2index = W1index + L*out2;
F1 = diag(.02*ones(1,10)); F2 = F1; k = 0.01;

%divide x into states
zeta = x(1:6); eta = x(7:12); psi = eta(1); th = eta(2); phi = eta(3);
intr1 = x(13:15); intr2 = x(16:18); V1 = reshape(x(19:V1index),in1,L);
V2 = reshape(x(V1index+1:V2index),in2,L);
W1 = reshape(x(V2index+1:W1index),L,out1);
W2 = reshape(x(W1index+1:W2index),L,out2);

% system dynamics
Teta = [-sin(th) 0 1; cos(th)*sin(psi) cos(psi) 0; cos(th)*cos(psi)...
    -sin(psi) 0];
Tetadot = [-cos(th)*eta(5) 0 0; (-sin(th)*eta(5)*sin(psi) + cos(psi)*eta(4)*cos(th))...
    -sin(psi)*eta(4) 0; (-sin(th)*eta(5)*cos(psi) - sin(psi)*eta(4)*cos(th))...
    -cos(psi)*eta(4) 0];
Sigma = diag([5,5,5]); J = Teta'*Sigma*Teta;
Axyz = [zeta(4)^2 * eta(6); zeta(5)^2 * eta(5); zeta(6)^2 * eta(4)];
Aeta = [zeta(4)*eta(6)^2; zeta(5)*eta(5)^2; zeta(6)*eta(4)^2];
c11 = (2*Sigma(2,2)*cos(th)^(2)*sin(psi)*cos(psi) - 2*Sigma(3,3)*cos(th)^(2)...
    *cos(psi)*sin(psi))*eta(4) + ((Sigma(2,2)-Sigma(3,3))*cos(th)*...
    (cos(psi)^(2)-sin(psi)^(2)))*eta(5);
c12 = (2*Sigma(1,1)*sin(th)*cos(th) - 2*Sigma(2,2)*sin(psi)^(2)*sin(th)*cos(th)...
    - 2*Sigma(3,3)*sin(th)*cos(th)*cos(psi)^(2))*eta(4) - ((Sigma(2,2)-Sigma(3,3))...
    *sin(psi)*cos(psi)*sin(th))*eta(5) - Sigma(1,1)*cos(th)*eta(6);
c13 = 0;
c21 = ((Sigma(2,2)-Sigma(3,3))*cos(th)*(cos(psi)^(2)-sin(psi)^(2)))*eta(4)...
    + (-2*Sigma(2,2)*cos(psi)*sin(psi) + 2*Sigma(3,3)*sin(psi)*cos(psi))*eta(5);
c22 = (-(Sigma(2,2)-Sigma(3,3))*sin(th)*sin(psi)*cos(psi))*eta(4);
c23 = 0; c31 = 0; c32 = -Sigma(1,1)*cos(th)*eta(4); c33 = 0;
```

```
c = [c11 c12 c13; c21 c22 c23; c31 c32 c33];
C = 2*Teta'*Sigma*Tetadot - 0.5*c;

%desired trajectory
xd0 = 0; xdf = 20; tf = 60; dx = (xdf - xd0)/(tf^3);
yd0 = 0; ydf = 5; dy = (ydf - yd0)/(tf^3);
zd0 = 0; zdf = 10; dz = (zdf - zd0)/(tf^3);
DTM = [1 tf tf^2; 3 4*tf 5*tf^2; 6 12*tf 20*tf^2];
ax = DTM\[dx 0 0]'; ay = DTM\[dy 0 0]'; az = DTM\[dz 0 0]';
xd = ax'*[t^3 t^4 t^5]'; yd = ay'*[t^3 t^4 t^5]'; zd = az'*[t^3 t^4 t^5]';
xddot = ax'*[3*t^2 4*t^3 5*t^4]'; yddot = ay'*[3*t^2 4*t^3 5*t^4]';
zddot = az'*[3*t^2 4*t^3 5*t^4]';
zetad = [xd; yd; zd; xddot; yddot; zddot];
xdddot = ax'*[6*t 12*t^2 20*t^3]'; ydddot = ay'*[6*t 12*t^2 20*t^3]';
zdddot = az'*[6*t 12*t^2 20*t^3]';
zetadddot = [xdddot; ydddot; zdddot];
psid = 0;

%control
e1 = zetad(1:3) - zeta(1:3); e1dot = zetad(4:6) - zeta(4:6);
r1 = e1dot + Lamb1*e1; X1 = [zeta(1:3); r1; eta(1:6)];
mu1 = (1./(1+exp(-V1'*X1)));
y1 = W1'*mu1;
Fhatd = m*zetadddot - m*Lamb1^(2)*e1 - [0; 0; -m*g] - y1 + Kr1*r1 +...
    Ki1*intr1;
a = -Fhatd(1); b = (Fhatd(2)^(2)+Fhatd(3)^(2))^(1/2);
ud = (a^2 + b^2)^(1/2); thd = atan(a/b); phid = atan(Fhatd(2)/Fhatd(3));
e2 = [psid-psi; thd-th; phid-phi];
if (t-tOld) ~= 0
    e2dot = (e2 - e2old)/(t - tOld);
    r2 = e2dot + Lamb2*e2;
else
    e2dot = e2dotold;
    r2 = e2dot + Lamb2*e2;
end
X2 = [eta(1:3); r1; r2];
mu2 = (1./(1+exp(-V2'*X2)));
y2 = W2'*mu2;
tau = y2 + Kr2*r2 + Ki2*intr2;
e2old = e2; tOld = t; e2dotold = e2dot;
r = [r1;r2];

%derivative calculation
zetadot = [zeta(4); zeta(5); zeta(6); (-ud/m)*sin(th) + (1/m)*Axyz(1);...
    (ud/m)*cos(th)*sin(phi) + (1/m)*Axyz(2); (ud/m)*cos(th)*cos(phi) - g + (1/m)*Axyz↙
(3)];
etadot = [eta(4); eta(5); eta(6); J\(tau - C*[eta(4); eta(5); eta(6)] + Aeta)];
V1dot = zeros(L*in1,1); V2dot = zeros(L*in2,1);
W1dot = F1*mu1*r1'-k*norm(r)*F1*W1; W2dot = F2*mu2*r2'-k*norm(r)*F2*W2;
xdot = [zetadot; etadot; r1; r2; V1dot; V2dot;reshape(W1dot,L*out1,1);reshape(W2dot,↙
L*out2,1)];
```

```
end
>> type calculateControl

function [ud,tau] = calculateControl(t,x)
%Recalculate control
%Quadcopter dynamics for nonlinear control final
global e2old tOld e2dotold
%constants
g = 9.81; L = 10; in1 = 12; in2 = 9; out1 = 3; out2 = 3; m = 1;
Lamb1 = diag(1*ones(1,3)); Lamb2 = diag(1*ones(1,3)); Kr1 = diag(15*ones(1,3));
Kr2 = diag(15*ones(1,3)); Ki1 = diag(5*ones(1,3)); Ki2 = diag(5*ones(1,3));
V1index = 19+L*in1-1; V2index = V1index + L*in2;
W1index = V2index + L*out1;
W2index = W1index + L*out2;

%divide x into states
zeta = x(1:6); eta = x(7:12); psi = eta(1); th = eta(2); phi = eta(3);
intr1 = x(13:15); intr2 = x(16:18); V1 = reshape(x(19:V1index),in1,L);
V2 = reshape(x(V1index+1:V2index),in2,L);
W1 = reshape(x(V2index+1:W1index),L,out1);
W2 = reshape(x(W1index+1:W2index),L,out2);

%desired trajectory
xd0 = 0; xdf = 20; tf = 60; dx = (xdf - xd0)/(tf^3);
yd0 = 0; ydf = 5; dy = (ydf - yd0)/(tf^3);
zd0 = 0; zdf = 10; dz = (zdf - zd0)/(tf^3);
DTM = [1 tf tf^2; 3 4*tf 5*tf^2; 6 12*tf 20*tf^2];
ax = DTM\[dx 0 0]'; ay = DTM\[dy 0 0]'; az = DTM\[dz 0 0]';
xd = ax'*[t^3 t^4 t^5]'; yd = ay'*[t^3 t^4 t^5]'; zd = az'*[t^3 t^4 t^5]';
xddot = ax'*[3*t^2 4*t^3 5*t^4]'; yddot = ay'*[3*t^2 4*t^3 5*t^4]';
zddot = az'*[3*t^2 4*t^3 5*t^4]';
zetad = [xd; yd; zd; xddot; yddot; zddot];
xdddot = ax'*[6*t 12*t^2 20*t^3]'; ydddot = ay'*[6*t 12*t^2 20*t^3]';
zdddot = az'*[6*t 12*t^2 20*t^3]';
zetadddot = [xdddot; ydddot; zdddot];
psid = 0;

%control
e1 = zetad(1:3) - zeta(1:3); e1dot = zetad(4:6) - zeta(4:6);
r1 = e1dot + Lamb1*e1; X1 = [zeta(1:3); r1; eta(1:6)];
mu1 = (1./(1+exp(-V1'*X1)));
y1 = W1'*mu1;
Fhatd = m*zetadddot - m*Lamb1^(2)*e1 - [0; 0; -m*g] - y1 + Kr1*r1 +...
    Ki1*intr1;
a = -Fhatd(1); b = (Fhatd(2)^(2)+Fhatd(3)^(2))^(1/2);
ud = (a^2 + b^2)^(1/2); thd = atan(a/b); phid = atan(Fhatd(2)/Fhatd(3));
e2 = [psid-psi; thd-th; phid-phi];
if (t-tOld) ~= 0
    e2dot = (e2 - e2old)/(t - tOld);
    r2 = e2dot + Lamb2*e2;
else
```

```
    e2dot = e2dotold;
    r2 = e2dot + Lamb2*e2;
end
X2 = [eta(1:3); r1; r2];
mu2 = (1./(1+exp(-V2'*X2)));
y2 = W2'*mu2;
tau = y2 + Kr2*r2 + Ki2*intr2;
e2old = e2; tOld = t; e2dotold = e2dot;
end
>>
```