

A Neural Autopilot Training Platform based on a Matlab and X-Plane co-simulation

Jérémy Pinguet, Philippe Feyel, Guillaume Sandou

▶ **To cite this version:**

Jérémy Pinguet, Philippe Feyel, Guillaume Sandou. A Neural Autopilot Training Platform based on a Matlab and X-Plane co-simulation. International Conference on Unmanned Aircraft Systems, Jun 2021, Athens, Greece. 10.1109/icuas51884.2021.9476679 . hal-03943947

HAL Id: hal-03943947

https://centralesupelec.hal.science/hal-03943947

Submitted on 17 Jan 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Neural Autopilot Training Platform based on a Matlab and X-Plane co-simulation

Jérémy Pinguet[1,2], Philippe Feyel[1], and Guillaume Sandou[2]

***Abstract*— The main objective of this paper is to describe a tool for the aircraft autopilot deployment only based on a flight database. Flight simulators such as X-Plane turn out to be powerful and efficient tools for creating such database and controller experimentation. The paper outlines the development of a co-simulation framework between Matlab and X-Plane using the User Datagram Protocol (UDP). The flight data collected during a first step are then used for the training of neural controllers. The approach is based on the neural network imitation ability to learn the piloting skills implicitly stored in the dataset. Also, in order to include fault-tolerant control, a Neural Multiple Model Adaptive Control (NMMAC) based on previously learned networks is implemented. This architecture consists of a bank of local controllers and a switching logic using a bank of estimators. As an illustration of the proposed platform, it is assumed that the airspeed is unmeasured for the flight director. A neural guidance autopilot based NMMAC is therefore performed on different airspeed values. Experiments show that the designed neural autopilot can successfully track both heading and altitude reference signals, while the method is not restricted to this scope.**

I. Introduction

In the challenging field of autonomous aircraft flight, the modern autopilot has proven to be very efficient for low-level piloting tasks in most flight situations. Still, the occurrence of unusual cases can lead the autopilot to disengage or, in more critical cases, to execute undesired actions that can compromise flight safety, as mentioned in [1]. For instance, strong turbulence or an aircraft position at the stall limit will cause the autopilot to disengage, and the pilot then has to take control of the aircraft. Due to the limited autopilots capabilities, the crew or the pilot must continuously monitor the flight status and quickly rectify in case of emergencies or unexpected situations. Overcoming these limitations by considering a set of rules to be respected through specific control modes seems unfeasible because of the high number of specifications and the complexity to formulate them mathematically. On the other hand, pilots are trained to handle difficult situations and flight uncertainties. Thus, an alternative method for autopilot design is to record the action performed by a pilot for a given state of the system and certain external conditions. In such a case, these decisions indirectly translate the specifications, and so the controller can be trained on the database gathering this information which leads to an intelligent controller that can manage either nominal flights or emergency situations. The purpose of the autopilot will be to act as an adviser to the pilot before taking control of the aircraft in the long term.

The Neural Autopilot Training Platform (NATP) is divided into three main modules corresponding to the three steps in the development and implementation of the autopilot:

- Collection of pilot data: During different periods of flight, a database gathers flight data coupled with the actions of a real pilot or an already implemented autopilot.
- Offline learning: A neural controller is trained on the database, and system training is also useful for developing the autopilot.
- Autonomous control: The online implementation of the autopilot is established alongside the aircraft simulation. The aircraft supplies the relevant data (position, attitude, etc.) to the autopilot so that it can calculate and send back the signal control values (actuators position, reference guidance angle, etc.).

This work aims to describe a controller design tool that can learn from a flight database. The major contributions lie in the formalization of a complete method from data acquisition to autopilot implementation in three steps that are often addressed separately in the literature. The NATP is organized into three modules as shown by the flowchart in Fig 1. The modules can be of interest to the reader independently, and the sections can be read separately.

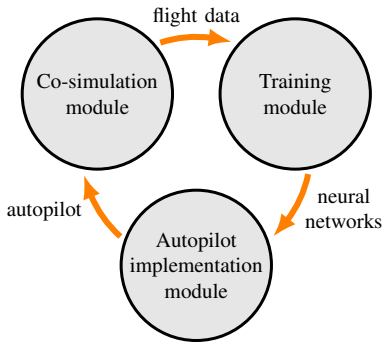


Fig. 1: The three modules of the Neural Autopilot Training Platform

Firstly, a co-simulation module is realized for data collection and aircraft simulation. Due to the recent significant progress in the field, flight simulators offer a high degree of realism, and thus they are used as training cockpits for pilots. These software programs are also attractive from the engineer perspective for their accuracy in aerodynamic calculations

[1]Research & Technology department, Safran Electronics & Defense, 91300, Massy, France, E-mail: {jeremy.pinguet, philippe.feyel}@safrangroup.com

[2] Université Paris-Saclay, CNRS, CentraleSupélec, Laboratoire des signaux et systèmes, 91190, Gif-sur-Yvette, France, E-mail: guillaume.sandou@centralesupelec.fr

and their ability to exchange data with external systems. X-Plane is one of the most popular realistic flight simulations software and has already been successfully used in many research projects on Unmanned Aerial Vehicle (UAV), such as UAV dynamics identification [2], UAV structure development [3] or control law validation [4], [5]. It offers a wide range of simulation diversity since its use as a simulation brick for the development of controllers has been realized for other various aircrafts such as helicopters [6], quadrotors [7], flapping-wing UAV [8], or tilt-rotor UAV [9]. Data transmission with X-Plane is ensured by the User Datagram Protocol (UDP), which is a relatively simple communication protocol, so the advantages and drawbacks lie in the absence of mechanisms like handshaking, back confirmation or Cyclic Redundancy Check (CRC) checksum. So the flight simulator is an opportunity to collect flight data, including the piloting skills of a human pilot or an implemented and approved autopilot. This paper described the development of a co-simulation module, which is running X-Plane in co-simulation with Matlab & Simulink. The simulator models the aircraft dynamics while Matlab will compute the autopilot control system in the end.

Secondly, a training module takes advantage of the flight data to learn the controller. The machine learning methods have a wide range of flexible tools that can learn these types of complex input-output mappings. The complexity of the processes to be identified leads to the use of neural network controllers for their universal approximation property [10]. In an imitation approach using pilot experience, supervised training techniques have proven performance in autopilot [1], [11], [12] or high-level controller such as navigation [13]. In this work, a framework is proposed with neural networks for control, and more precisely an identification method based on Outputs State-Space Neural Network (OSSNN) is presented.

Thirdly, autonomous control is finally carried out by the autopilot implementation module. The multiple model approach provides a powerful solution to many problems dealing with complex real-life processes based on the problem decomposition strategy. The Multiple Models Adaptive Control (MMAC) assumes that the system to be controlled can be represented by a finite number of sub-models, each of them being valid in a given operating domain. Some controllers are respectively designed for each model, and a selection algorithm decides at each moment which controller(s) should be involved for the real system. MMAC is a widely used control strategy in the context of autonomous aircraft flight, such as in [14], [15], [16], [17]. In such circumstances, each local model describes a particular fault scenario or operating condition. The study presents a fault-tolerant control denoted Neural MMAC (NMMAC), which combines a multiple models approach with pre-trained neural networks.

This paper is organized as follows: section II presents the co-simulation framework. Neural networks training is described in section III and the NMMAC is the subject of section IV. Experiments results that combine the presented techniques, are given in section V. Section VI provides some conclusion while section VII offers some forthcoming works.

II. The co-simulation module

In the co-simulation approach, the flight dynamics calculations are performed by the simulator. This process works as a black box for Matlab as it would be the case for a real aircraft (Fig. 2). This method benefits from the high degree of realism provided by the simulator, which uses realistic flight dynamics models. The experiment platform is based on X-Plane 11, a commercially flight simulator developed by Laminar Research. This software was used in this work because of its high simulation reliability and its flexibility of use. This section highlights the key steps to develop the co-simulation environment with Matlab R2019b using the *Instrument Control Toolbox* and the *Simulink Desktop Real-Time Toolbox*. Because of the required computational resources and real-time simulation constraint, both software programs are run on two separated computers.

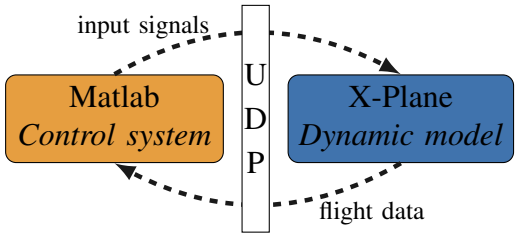


Fig. 2: Co-simulation between X-Plane and Maltab

A. X-Plane presentation

Thanks to its extreme accuracy, X-Plane enables a simulation with a high degree of similarity compared with an actual flight. This simulator developed as an engineering tool is distinguished by its flight model based on the aircraft geometric shape. The simulation models the forces acting on the different parts of the aircraft in real-time to determine lift and drag, and then calculates the acceleration, speed and position of the aircraft. This process called blade element theory involves breaking a blade down into several small parts, then determining the forces on each of these small blade elements. This approach differs from the conventionally used method of aggregating empirical measures and determining aerodynamic forces with predefined lookup tables. The X-Plane simulator is certified by the Federal Aviation Administration (FAA) to be used by pilots as a training tool while connecting with certified hardware (cockpit and flight controls).

Another key element of X-Plane is its wide range of options allowing the creation of a rich simulation environment. It is possible to easily use different aircrafts, including prototype models designed with the Plane-Maker. Many settings on the aircraft allow simulating failure scenarios or other specific events that are very useful for intelligent autopilots training. Other features are also integrated with X-Plane for environmental issues management, such as the atmospheric conditions, wind speed and direction or turbulence. Moreover, the simulator game interface enables a pilot intervention to analyze his behaviour and record his reaction in various situations.

B. Data interface

Communication between the simulator and Matlab is ensured by the UDP communication. This telecommunication protocol is part of the transport layer of the Open Systems Interconnection (OSI) model. It allows the transmission of data in the form of datagrams in a simple way between two entities defined by their IP address and a port number. This approach makes it possible to run the two softwares on different platforms as long as they are on the same local network. This protocol does not have a handshaking mechanism, and it is exposed to network reliability problems. Consequently, transmission is not guaranteed in terms of delivery and accuracy of data such as the order of arrival or the possible duplication of datagrams. This protocol is therefore, suitable for use without detection and correction of errors. In a real-time simulation context, the UDP nature makes it useful for quickly transmitting small amounts of data, since it is preferable to eventually lose a datagram rather than wait for it to be retransmitted.

X-Plane data packet follows a standard format consisting of stream of datasets. The data package is a string of bytes illustrated by Table I and respecting the following pattern:

- Header consisting of 5 bytes: The first four bytes are ASCII code for the key words 'DATA' indicating that it is a data packet. The fifth byte is an internal code defined as '0'.
- Dataset consisting of 36 bytes repeated by the total number of datasets: The four bytes following the header are the dataset index in decimal from 0 to 133. The next 32 bytes represent the data of 8 parameters with 4 bytes for each. One parameter representing the data value in single precision floating point. The other datasets follow by packets of 36 bytes respecting the same pattern. As a format illustration, the Table II gives an example of dataset.

Other clarifications may be added to facilitate implementation:

- The length of the data packet varies according to the total number of selected datasets. According to their index ascending order, datasets are then sorted one after the other in the message. The dataset with the lowest index will be inserted just after the header and so on.
- A majority of datasets contain less than 8 parameters, therefore the unused information is filled by the number '-999'. This number, when used for sending, also means that the parameter value will not be changed in X-Plane.

TABLE I: X-Plane data packet format

Header	Dataset n°1		Dataset n°2		...
5 bytes	9x4 bytes		9x4 bytes		
'DATA0'	Index	8 Parameters	Index	8 Parameters	...
5 bytes	4 bytes	8x4 bytes	4 bytes	8x4 bytes	

C. X-Plane settings

The development of X-Plane as an engineering tool makes its use very flexible for experimentation. Many data are accessible and can be displayed during simulation, sent over a network, or written to a file. Data selection and use are made through the data output table. The table indicates datasets indexes for proper identification, however their contents can be obtained from their display in the cockpit or the record file. Users can choose the datasets to be sent or saved and the rate up to 99.9 data packets per second. For the UDP correct operation, the machine IP address where Matlab is running must be specified likewise as the receiving port to bind.

TABLE II: Dataset format example

	"Pitch, roll & headings" Dataset				
	Index	Parameter 1	Parameter 2	...	Parameter 8
Labels	17	pitch, deg	roll, deg	...	none
Bytes	17-0-0-0	156-162-9-192	240-92-19-65	...	0-192-121-196
Values	$2.3822\ 10^{-44}$	–2.15055	9.21016	...	-999

Many other options are also available, especially in terms of graphics. The display level can be adjusted to suit the machine used to ensure the real-time performance of the simulator. A parameter that can be important is the flight model per frame. This parameter represents the number of times the simulator calculates the aerodynamic forces for each frame. It must be adjusted if the frame rate is very low and even more so if the plane is small, fast or light.

D. Matlab & Simulink settings

On Matlab side, receiving and sending data to X-Plane is handled in the Simulink environment. The *Instrument Control Toolbox* provides blocks for communicating over UDP network. It is possible to bypass the use of the toolbox by using a similar method to the one proposed in [18]. Nevertheless, attention must be paid to the code optimization and its computation speed not to slow down the execution of Matlab and its good real-time operation.

Receiving and decrypting data in the Simulink environment involves the following steps:

- *UDP Receive*: Receive the UDP message from the IP address where X-Plane software runs and specifies the port to bind. The received message is in 8-bit unsigned integer variable (uint8) format and its size will be 5 bytes for the header plus 36 bytes per each selected output dataset in the X-Plane data output table.
- *Byte Unpack*: Unpack data into an array of bytes. The header is an uint8 equal to '68-65-84-65-48' for 'DATA0' whereas the rest of the message must be separated into 9 groups of single-precision variables (32-bit) for each dataset.
- *MATLAB Function*: Obtain each parameter individually by selecting the dataset in which it is contained and its position within that dataset.
- *Data Type Conversion*: Convert data in double-precision variables is often required for correct use with Simulink.

Other blocks handle the data transmission to the simulator. The number and selection of datasets sent are independent

of those received. A similar reverse protocol is used to send the desired data to X-Plane:

- *Data Type Conversion*: Convert data in single-precision variable.
- *MATLAB Function*: Create the datasets that need to be modified by placing the 8 parameters in the right order and adding the value '-999' in uint8 format for unused or unmodified parameters. The index of the corresponding dataset must precede each group of parameters also in uint8 format.
- *Byte Pack*: Pack input data into a single output vector of uint8 containing the header and datasets in succession without worrying about order.
- *UDP Send*: Send the UDP message to X-Plane with the correct IP address and a different port than the receiving one.

A crucial point of the co-simulation relies on time synchronization. Care must be taken that Matlab does not take to much time to compute and send data to X-Plane. Otherwise packages are buffered and Matlab calculations are based on out-dated flight data, leading to unintended behavior. To our knowledge, there is no possibility to synchronize the clocks of the flight simulation software and Matlab. Even if it is possible to imagine a pause and play mode sequencing of X-Plane enforced by Matlab, this manipulation turns out to be impractical. The simplest way to proceed is to set a real-time operation on both software. On the X-Plane side, the simulation follows the real-time performance, and it is very straightforward to spot a simulation slowdown through warning messages. On Matlab the use of the *Real-Time Synchronization* block provide by the *Simulink Desktop Real-Time Toolbox* allows to run the simulink model with a real-time clock.

III. The training module of neural networks for control

Black-box identification of processes based on input-output data is performed with neural networks. The use of specific shallow network structures provides a link with control theory, and a training method is then developed on Matlab with the help of the *Deep Learning Toolbox*.

A. Training procedure

The learning considered in this work does not require any knowledge of the internal structure of the process to be identified. Black-box training focuses on stimuli inputs and output reactions. Thus, the unknown system is based on a finite database consisting of pairs of input-output vectors $\mathrm{Z_{dt}} = \{\mathrm{u_{dt}(k)}, \mathrm{y_{dt}(k)} \mid \mathrm{k} = 1,..., \mathrm{N_{dt}}\}$ with $\mathrm{u_{dt}(k)} \in \mathbb{R}^{\mathrm{m_u}}$ and $\mathrm{y_{dt}(k)} \in \mathbb{R}^{\mathrm{m_y}}$. The data are assumed to be rich enough and with a proper sampling frequency to characterize the system dynamics as well as the corresponding control law, and the noise is supposed to be centered. In practice, these data may be derived from an acquisition phase followed by a pre-processing phase (cleaning, reduction, partitioning, etc.). Multilayer perceptron (MLP) is one of the most widely used structures for system identification. The training method consists of supervised learning on the dataset, whose goal is to obtain a time series regression. To this end, weights and biases of the network are optimized to minimize a fitness function which is here the Mean Square Error (MSE) between the estimated outputs $\mathrm{y_{nn}}$ and the measured outputs $\mathrm{y_{dt}}$:

$$J = \frac{1}{2\mathrm{N_{dt}m_y}} \sum_{\mathrm{k}=1}^{\mathrm{N_{dt}}} \left[\mathrm{y_{dt}(k)} - \mathrm{y_{nn}(k)}\right]^T \left[\mathrm{y_{dt}(k)} - \mathrm{y_{nn}(k)}\right] \tag{1}$$

The error is then back-propagated through the network to obtain the required derivatives for the use of gradient descent algorithms. Among the numerous learning algorithms [19], the most well known are *Backpropagation*, *Momentum Backpropagation*, *Levenberg-Marquardt (LM)*, *BFGS*, or *Conjugate Gradient*. Some of these algorithms require the Jacobian calculation of the cost function, which has to be approximated when networks are recurrent ones by algorithms such as *Real-Time-Recurrent-Learning (RTRL)* or *BackPropagation-Through-Time (BPTT)* [19].

A significant problem with training is the ability of the neural network to generalize. The network learns the submitted training data, but it must also be able to interpolate or extrapolate to new situations. As the network learning capacity and complexity increase with the number of neurons, the structures tend to be oversized, and the training is subject to overfitting. A way used in this work to avoid overfitting is to detect it using the cross-validation method and early stop the training. The database is finally separated into a learning set, used by the optimization algorithm; a validation set, whose learning error will cause the algorithm to stop if it increases a given number of times successively; a test set, independent of the learning process, allowing to measure its generality.

Besides, a normalization phase is added to the data pre-processing. This step is often essential to the learning process proper functioning and allows a faster convergence of the gradient descent algorithm. The used normalizations are those known as *min-max* and *z-score*. The first is one of the simplest methods of resizing the data range between $[-1;\ 1]$, while the second imposes a null mean and a unit standard deviation on the data distribution.

Finally, the learning algorithms are relatively sensitive to initial points that will impact their more or less effective convergence towards the global optimum. It is a well-known procedure in the offline training context to make several runs and keep the one with the lowest cost.

B. Neural identification methods

Feedforward Neural Networks (FNNs) are inherently static structures. The way these networks can approximate spatio-temporal sequences is artificially realized by providing a series of past inputs and outputs to the network. Through tapped delay line (TDL), presented input-output vectors are $\mathrm{u(k)} = \left[\mathrm{u_{dt}^T(k)} \ \ \mathrm{u_{dt}^T(k\text{–}1)} \ \ \cdots \ \ \mathrm{u_{dt}^T(k\text{–}n_{in})}\right]^T \in \mathbb{R}^{\mathrm{n_u}}$ and $\mathrm{y(k)} = \left[\mathrm{y_{dt}^T(k)} \ \ \mathrm{y_{dt}^T(k\text{–}1)} \ \ \cdots \ \ \mathrm{y_{dt}^T(k\text{–}n_{out})}\right]^T \in \mathbb{R}^{\mathrm{n_y}}$ whose dimensions are $\mathrm{n_u} = \mathrm{m_u}(\mathrm{n_{in}} + 1)$,

$n_y = m_y(n_{out} + 1)$, so that the FNN can generate the estimated output:

$$y_{nn}(k+1) = f_{FNN}\left(\begin{bmatrix} u_{dt}(k) \\ \vdots \\ u_{dt}(k\text{–}n_{in}) \end{bmatrix}, \begin{bmatrix} y_{dt}(k) \\ \vdots \\ y_{dt}(k\text{–}n_{out}) \end{bmatrix}\right) \quad (2)$$

The main characteristic of these neural networks is that they do not have any intrinsic dynamics in their structure. The advantage of FNN training is that it is relatively quick and efficient. One of the well-known drawbacks of this approach is that it can only handle a finite number of past inputs and outputs. Furthermore, this representation suffers from high sensitivity to the delay windows and an unavoidable vulnerability to noisy inputs and outputs. Indeed, the learning stability of FNNs is not guaranteed, and the prediction can change significantly when the inputs are slightly modified. When FNNs are used as dynamic systems or long-term predictors, an output feedback loop is artificially created to provide the past outputs to the network. However, this feedback connection, which has not been taken into account during training, causes errors that may be further accumulated, which does not guarantee the use of FNNs as a dynamic model.

Another way to represent dynamic systems is to include feedback connections through Recurrent Neural Networks (RNNs):

$$y_{nn}(k+1) = f_{RNN}\left(\begin{bmatrix} u_{dt}(k) \\ \vdots \\ u_{dt}(k\text{–}n_{in}) \end{bmatrix}, \begin{bmatrix} y_{nn}(k) \\ \vdots \\ y_{nn}(k\text{–}n_{out}) \end{bmatrix}\right) \quad (3)$$

This recurrent structure embodies a larger class of nonlinear dynamical systems because the model dynamics intrinsically integrate the system history. Thus, RNNs provide universal identification models in the sense that they can uniformly approximate any nonlinear multi-inputs multi-outputs (MIMO) nonlinear dynamic system over a finite time interval, and for any continuous and bounded input signal [20], [21]. Moreover, RNNs are less vulnerable to noisy data because the input vector provided to the network does not include the previous outputs.

Despite being more stable in the learning meaning, RNNs have difficulty to converge during the training with data from highly complex systems. As previously mentioned, initial weights have an influential role in the convergence of learning algorithms. A proposed solution is to train the network with an FNN structure, then add artificial feedback to transform it as RNN to initialize the second training in RNN. This method significantly improves algorithm convergence and reduces the required time compared to classical RNN training.

A complementary method is used to enhance the FNN stability after training when used as closed-loop dynamic models. This method derived from the work of [22] consists of supervising a sliding window of outputs and not only outputs at a given moment. The learning fitness function, therefore, consists of minimizing the MSE between the estimated output window $\tilde{y}(k) = \left[y_{nn}^T(k)\ \ y_{nn}^T(k\text{–}1)\ \ \cdots\ \ y_{nn}^T(k\text{–}n_{out})\right]^T \in \mathbb{R}^{n_y}$ and the data window:

$$J_+ = \frac{1}{2N_{dt}n_y}\sum_{k=1}^{N_{dt}} [y(k) - \tilde{y}(k)]^T [y(k) - \tilde{y}(k)] \quad (4)$$

The resulting networks are found to accumulate fewer errors when they are closed into RNNs.

C. Outputs State-Space Neural Network (OSSNN)

The purpose here is to present a neural network topology than can efficiently represent a general class of nonlinear systems, and that can suit to control theory methods. The State-Space Neural Network (SSNN) allows using a nonlinear equation to represent the network dynamics in the classical control manner. There are numerous formulations and uses in the literature, as well as links established in particular with Linear Parameter-Varying (LPV) [23], [24], [25], [26]. The particular architecture proposed in this work is the Outputs State-Space Neural Network (OSSNN) consisting of one hidden layer, as depicted in Figure 3. This structure

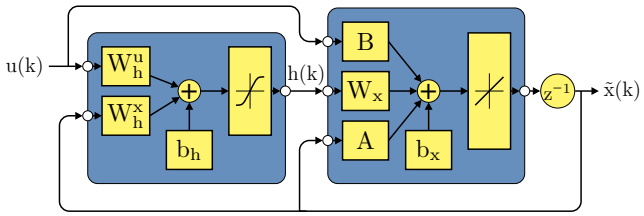


Fig. 3: OSSNN topology

allows representing the neural network in a state-space form whose state keeps a physical meaning since it is constituted of delayed outputs. This topology includes neurons with two types of activation functions, namely hyperbolic tangent activation and linear activation. The OSSNN have the following nonlinear state-space representation:

$$\begin{cases} h(k) &= \tanh(W_h^x \tilde{x}(k) + W_h^u u(k) + b_h) \\ \tilde{x}(k+1) &= A\tilde{x}(k) + Bu(k) + W_x h(k) + b_x \\ y_{nn}(k) &= C\tilde{x}(k) \end{cases} \quad (5)$$

where $\tilde{x}(k) = \left[y_{nn}^T(k)\ \ y_{nn}^T(k\text{–}1)\ \ \cdots\ \ y_{nn}^T(k\text{–}n_{out})\right]^T \in \mathbb{R}^{n_y}$ is the state vector, $u(k) \in \mathbb{R}^{n_u}$ is the input signal, $y_{nn}(k) \in \mathbb{R}^{m_y}$ is the output vector of the neural network, $W_h^u \in \mathbb{R}^{n_h \times n_u}$, $W_h^x \in \mathbb{R}^{n_h \times n_y}$ and $b_h \in \mathbb{R}^{n_h}$ are the weights and the bias of the hidden layer while $W_x \in \mathbb{R}^{n_y \times n_h}$, $A \in \mathbb{R}^{n_y \times n_y}$, $B \in \mathbb{R}^{n_y \times n_u}$ and $b_x \in \mathbb{R}^{n_y}$ are the weights and bias of the output layer. The activation function of the hidden layer is the hyperbolic tangent function. During the training phase, it is essential to set appropriate values for the network hyper-parameters, i.e. the number of neurons n_h in the hidden layer alongside with the size of the windows of past inputs n_{in} and outputs n_{out}.

The proposed OSSNN owns hybrid features since it includes a linear contribution from the linear activation function and a nonlinear contribution from the hidden layer activation function. The combination of both contributions improves the network modeling performance since dynamic systems also usually respect this hybrid behaviour.

While SSNNs are effective for dynamic system modeling, they can present the same training difficulties as other RNNs. Learning can be slow to converge and get stuck in local minima. An advantage of the OSSNN architecture is that through the physical meaning of the state, learning in FNN is possible. The state that consists of the delayed outputs is substituted in this learning type by the data output window to feed both network layers. The resulting network can then, as mentioned above, initiate a second training of the presented recurrent OSSNN. This structure, therefore, represents a practical compromise between easily trainable FNNs and very generic SSNNs.

IV. The Autopilot implementation module based on NMMAC

A. MMAC presentation

The multiple model approach is an adaptive control technique dealing with systems with large parametric uncertainties or with a strong nonlinear behaviour. The starting point is that a single fixed controller cannot stabilize all possible configurations and simultaneously meet some performance requirements. The plants are described by a combination of local models around operating points where each model is valid in a particular region.

Basically, MMAC architecture is divided into two autonomous entities. First, the control process consists of a bank of non-adaptive controllers designed for each model. The second entity is the adaptive mechanism composed of a bank of observers declined from models and the selection logic, which makes use of the monitoring outputs estimation errors. Typically, this second process employs the Multiple Model Adaptive Estimation (MMAE) consisting of a bank of Kalman Filters (KFs) (see [15], [27], [28]).

In the training platform framework, a nonlinear version of MMAC is involved, so that a nonlinear version of both subsystems is then presented in this section. The original MMAE is modified to an Extended MMAE by the use of Extended Kalman Filters (EKFs) as nonlinear estimators instead of the linear KFs (as in [15] or suggested in [29]).

B. MMAC architecture

Consider a plant model G subject to parameter variations $\theta \in \mathbb{R}^{n_\theta}$ taking values over a compact set $\Omega \subset \mathbb{R}^{n_\theta}$. It is assumed that G is a Multiple-Inputs-Multiple-Outputs (MIMO) plant model of the form:

$$G(\theta) := \left\{ \begin{array}{rcl} \dot{x}(t) & = & f\big(x(t), u(t), \theta(t)\big) \\ y(t) & = & g\big(x(t), \theta(t)\big) \end{array} \right. \tag{6}$$

where $x(t) \in \mathbb{R}^{n_x}$ denotes the state of the system, $u(t) \in \mathbb{R}^{n_u}$ is the vector of control inputs and $y(t) \in \mathbb{R}^{n_y}$ is the vector of measured outputs.

First let's suppose that N models are necessary to cover the uncertainty set of the real plant. Considering a finite set of candidate parameter values $\Theta := \{\Theta_1,...,\Theta_N\}$, for each nominal configuration $\Theta_i \in \Theta, i = \{1,...,N\}$ a corresponding nominal model is obtained $M_i := G(\Theta_i)$. For instance, Θ_i can represent a brutal variation in a physical parameter value due to default or a specific value of an unmeasured parameter, etc.

Once a bank of models has been obtained after a training procedure or by using physical equations, a bank of controllers can be associated with it. For each i^{th} local model, a local controller is designed to guarantee local stability and performance robustness. The bank of controllers is composed of N controllers K_i which have the following state-space representations:

$$K_i := \left\{ \begin{array}{rcl} \dot{x}_i^K(t) & = & f^K\big(x_i^K(t), r(t) - y(t)\big) \\ u_i(t) & = & g^K\big(x_i^K(t), r(t) - y(t)\big) \end{array} \right. \tag{7}$$

where $r(t) \in \mathbb{R}^{n_y}$ is the reference to be tracked. These controllers are partially combined to form the global control law. The final control signal applied to the real plant is a weighted sum of the outputs of each local controller:

$$u(t) = \sum_{i=1}^{N} w_i(t) u_i(t) \tag{8}$$

where $w_i(t)$ is called the validity reflecting the relative importance of each i^{th} model compared to the real plant. Suitable validities are assigned based on the probabilities such that less probable models are associated to smaller weights. This ensures that designed controllers for less probable models have less influence on the resulting control value. The validity vector $w \in \mathbb{R}^N$ have the following convex property:

$$\forall i \in \{1,...,N\}\ w_i(t) \geq 0\ , \qquad \sum_{i=1}^{N} w_i(t) = 1 \tag{9}$$

The next step is to determine a bank of estimators consisting of N estimators E_i that generate estimated system outputs $\hat{y}_i(t)$ for each nominal model M_i based on the applied control inputs $u(t)$ and measured outputs $y(t)$ of the real plant. These estimates will provide the basis for the validities computation. As mentioned above, EKFs, that are a nonlinear derivation of the original KFs, are used.

Fig.4 presents a general MMAC where $\hat{Y} = \left[\hat{y}_1^T \cdots \hat{y}_N^T\right]^T$ is the concatenation of the estimated outputs from the bank of estimators and $U = \left[u_1^T \cdots u_N^T\right]^T$ are the outputs of the bank of controllers.

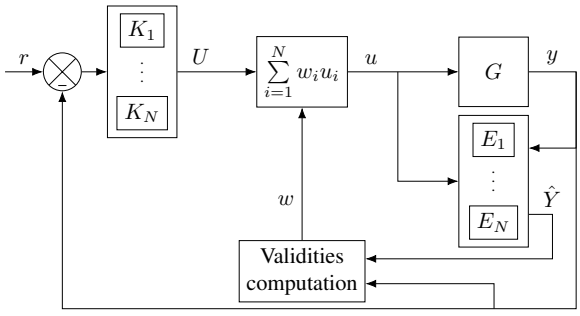


Fig. 4: MMAC method

The last and one of the most fundamental aspects of the MMAC is the methodology to determine the validity associated with each controller for the evaluation of the actual

control signals. This logic must identify which controllers must be involved in the control loop and their contributions depending on the real value of parameter θ that is supposed to be unknown. The validity calculation is based on the distance between the system output and those of the different estimators defined in the bank. This distance called residual or innovation is therefore defined for the i^{th} model as:

$$\varepsilon_i(t) = y(t) - \hat{y}_i(t) \tag{10}$$

The normalized validity is online-computed at each sample time index $k \in \mathbb{N}$ by the Posterior Probability Evaluator (PPE) in three step:

1) Recursive update:

$$w_i'(k) = \frac{\beta_i(k) e^{m_i(k)}\, w_i'(k\text{-}1)}{\sum_{j=1}^{M} \beta_j(k) e^{m_j(k)}\, w_j'(k\text{-}1)} \tag{11}$$

with

$$\beta_i(k) = \left[(2\pi)^{n_y} \det(S_i(k))\right]^{-\frac{1}{2}}$$
$$m_i(k) = -\tfrac{1}{2}\varepsilon_i^T(k) S_i^{-1}(k) \varepsilon_i(k)$$

2) Bounding away from zero:

$$w_i'(k) = \begin{cases} w_i'(k) & \text{if} \quad w_i'(k) > \delta \\ \delta & \text{otherwise} \end{cases} \tag{12}$$

3) Normalization:

$$w_i(k) = \begin{cases} \frac{w_i'(k)}{\sum_{j\in\mathcal{J}} w_j'(k)} & \text{if} \quad w_i'(k) > \delta \\ 0 & \text{otherwise} \end{cases} \tag{13}$$

where $\mathcal{J} := \{j \in \{1,...,N\} \mid w_j'(k) > \delta\}$

Validities are led by a Bayesian validity formula where $S_i(k)$ is the residual covariance matrix calculated by estimators. These matrices play an important role in tuning the convergence rate of probabilities $w_i(k)$. To keep models alive, a threshold δ is used. The probabilities are bounded to this value and not allowed to go to zero. This threshold represents the sensitivity of the algorithm to new information, a high value will improve responsiveness while a lower value will yield less varying probabilities. At last, the weights are normalized by excluding the models having reached the threshold.

C. Neural MMAC (NMMAC)

In the proposed flight data learning approach, the blocks that constitute the MMAC are based on the already trained neural networks. The data is separated according to different aircraft operating points, that are particular values of varying parameters or specific failure scenarios.

The networks structure is the OSSNN presented in the previous section. The neural controllers identified on the data can be directly used to build the bank of controllers. On the other hand, the state-space representation of the OSSNN allows the implementation of nonlinear observers such as EKFs to form the bank of observers.

V. Experiments

The experiment presented here is intended to illustrate the proposed NATP rather than a practical case. The interest of this example is solely to demonstrate the platform capabilities through a simple application of the intelligent controller built with the NATP. Thus, it could be more sophisticated in the long term and deal with more realistic problems, for instance, the reconfiguration of low-level controllers in case of default or an intelligent controller that could be trained to steer several planes.

The illustration of the NATP is based on an aircraft guidance controller, also named flight director, whose airspeed is not measured for control purposes. Therefore, an NMMAC considering longitudinal and lateral coupled dynamics is performed on three different airspeed values selected inside the aircraft flight envelope. The simulations reported in this paper use the Vision SF50. It is a single-engine, very light jet aircraft manufactured by Cirrus Aircraft. This model, which is one of those installed in the X-Plane default fleet, is equipped with an autopilot. The objective is to use the flight data to recreate this autopilot in a neural version organized in an NMMAC, as shown in Fig 5. Remember that a real recorded flight could also generate those data.

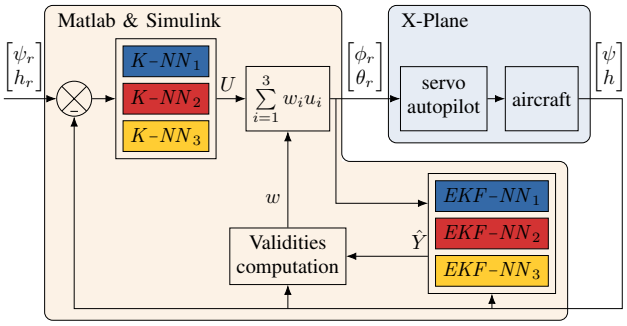


Fig. 5: Neural autopilot

Black-box models of the aircraft and the existing corresponding autopilot are identified by neural networks for the three airspeed values $\Theta = \{v_1, v_2, v_3\}$, leading to the neural controllers $K\text{-}NN_i$ and the EKFs based on neural models $EKF\text{-}NN_i$. The controllers are those of guidance, and the identified systems include the aircraft dynamics together with the internal servo autopilot of X-Plane. The neural autopilot aims to track the heading ψ_r and altitude h_r reference signals. The control signals composed by the roll reference ϕ_r and pitch reference θ_r are computed in Matlab and sent to the simulator, which returns the aircraft heading ψ and altitude h.

A. Data collection

In an effort to have a sufficiently rich learning signal, heading and altitude references are sent to X-Plane, and the autopilot has the task of controlling the aircraft in complying with these references. These reference signals are composed of a succession of steps with random amplitude and length. Amplitudes are bounded by minimum and maximum values,

which are forced to appear in the excitation signal at least once to ensure the system is excited on the desired range. Similarly, steps full duration are determined so that the steady-state is reached in a more or less important proportion of the cases. The desired headings are chosen between 150° and 300°, and the steps have a duration not exceeding 60s. Simultaneously, the specified altitude steps are bounded by 3500 feet and 6000 feet and their duration by 120s. During altitude change phases, the autopilot manages the rate of climb and maintains a vertical speed of 1500 ft/min.

During the flight, the autopilot also maintains airspeed (Knots Indicated Airspeed) around three successive operating points, $v_1 = 130$kt, $v_2 = 160$kt and $v_3 = 190$kt. Each of the flight phases is 45min long, X-Plane records the data at a writing frequency of 20Hz, i.e. a sampling time of 0.05s which is a typical guidance frequency. The flight is carried out in calm weather around Paris Orly airport.

B. Neural networks training

Once the flight data is available, it is extracted from the text file and separated according to the three operating points for the offline learning in Matlab. In the pre-processing, each dataset is resampled at 0.05s for a uniform sampling time, normalized using the *min-max* method before being divided into three sets with 70% for the training set, 15% for the validation set and 15% for the test set.

The used learning algorithm is the LM which is usually the most efficient for shallow network training, combined with RTRL for the Jacobian approximation during RNN training. The early stopping criterion is used when the validation cost increases 300 times successively. A number of 8 runs is performed, and the network with the lowest validation performance is selected. Finally, the input and output weights are modified according to the data normalization values to avoid input-output normalization and de-normalization when using the network. Each run is divided into two steps: the network is first trained in FNN, i.e. the outputs vector is supplied to the network; the obtained network is then used to initiate a second RNN training where generated outputs are fed back to the network. An OSSNN structure is used, a succession of trial-and-error tests are required to set the hyper-parameters $\mathrm{n_h}$, $\mathrm{n_{in}}$ and $\mathrm{n_{out}}$ to obtain the most optimized networks.

Table III and Table IV respectively show the best results for identifying the systems and the controllers on the three-speed operating points $v = \{130\text{kt}, 160\text{kt}, 190\text{kt}\}$. The hyper-parameters used for each training are specified in the table title. The given results are: the performance according to the MSE (4) obtained on each data set, the number of iterations where the minimum is reached before the validation cost increases, and the computing time required by parallelizing the calculations on an Intel(R) Core(TM) i5-8400H-2.50GHz processor and Matlab 2019b.

C. Autonomous control

The NMMAC is now built based on the trained neural networks. The controllers are directly used in their neural

TABLE III: System training results on the input-output data $\left\{\begin{bmatrix}\phi_r & \theta_r\end{bmatrix}^T, \begin{bmatrix}\psi & h\end{bmatrix}^T\right\}$ with $\mathrm{n_h} = 15$, $\mathrm{n_{in}} = 3$ and $\mathrm{n_{out}} = 1$

	training type	training perf.	validation perf.	test perf.	iterations	time
Model n°1	FNN	3.29e−12	5.44e−12	7.53e−12	388	2m12s
	RNN	5.65e−4	4.93e−3	3.51e−2	303	11m16s
Model n°2	FNN	7.39e−12	8.43e−12	1.72e−11	345	1m46s
	RNN	3.60e−4	1.01e−3	2.79e−2	302	11m12s
Model n°3	FNN	5.94e−11	3.55e−11	6.97e−11	423	1m18s
	RNN	8.03e−6	3.73e−4	5.94e−3	780	18m20s

TABLE IV: Controller training results on the input-output data $\left\{\begin{bmatrix}\psi_r\text{-}\psi & h_r\text{-}h\end{bmatrix}^T, \begin{bmatrix}\phi_r & \theta_r\end{bmatrix}^T\right\}$ with $\mathrm{n_h} = 15$, $\mathrm{n_{in}} = 3$ and $\mathrm{n_{out}} = 1$

	training type	training perf.	validation perf.	test perf.	iterations	time
Controller n°1	FNN	2.20e−7	2.54e−7	2.43e−7	212	13s
	RNN	7.09e−3	4.70e−4	9.77e−3	302	3m22s
Controller n°2	FNN	2.47e−7	2.90e−7	1.81e−7	231	18s
	RNN	6.83e−3	7.11e−3	8.44e−3	159	1m56s
Controller n°3	FNN	2.85e−7	2.34e−7	2.71e−7	1111	52s
	RNN	9.62e−3	7.12e−3	1.32e−2	20	10s

form to constitute the bank of controllers. Discrete EKFs are designed from the OSSNNs to form the bank of observers. EKFs are nonlinear observers that can be easily implemented, but their design can be challenging to set up. A first setting is obtained by using the MMAE alone and by feeding the MMAE with flight data. EKFs are then fine-tuned when using the MMAC with X-Plane, and the following setting is retained for each i^{th} estimator: the process noise covariance $Q_i = diag(\begin{bmatrix}10^{-4} & 10^{-3} & 10^{-4} & 10^{-3}\end{bmatrix})$, the observation noise covariance $R_i = diag(\begin{bmatrix}1 & 10\end{bmatrix})$ and the initial estimate covariance $P_i(k{=}0) = 10^3$. The residuals covariance S_i are then used by the PPE to calculate the validities with the following settings: a threshold $\delta = 0.01$ and initial weights $w_i(k{=}0) = 1/3$.

The control system is then co-simulated with X-Plane, as previously shown in Fig 5. The proposed neural autopilot tracks two reference signals, including steps and ramps. The airspeed is managed by the inner servo autopilot, but it is not taken into account by the neural autopilot in Matlab. Fig 6 shows the results obtained for a flight in calm weather, and Fig 7 shows the comparison of the responses with the imitated X-Plane autopilot. The results show satisfying performance in controlling heading and altitude that match X-Plane autopilot, appropriate control signals followed by the internal aircraft autopilot, and convergence of the validities according to the unknown speed according to the neural autopilot.

VI. Conclusions

The research principal goal was to develop a platform to train a neural autopilot to mimic the piloting abilities of an artificial or human pilot known to be skilled in handling real-world flight conditions. For the purpose set, a controller

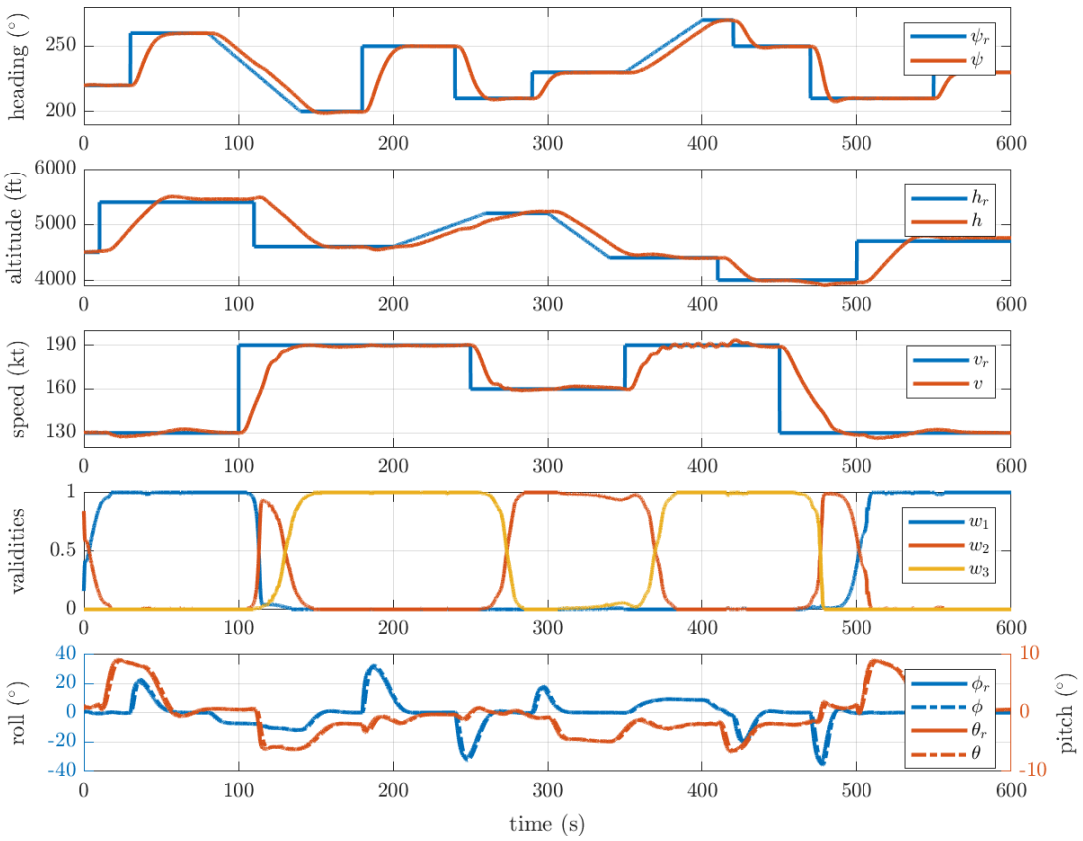


Fig. 6: Proposed NMMAC responses for a airspeed variations

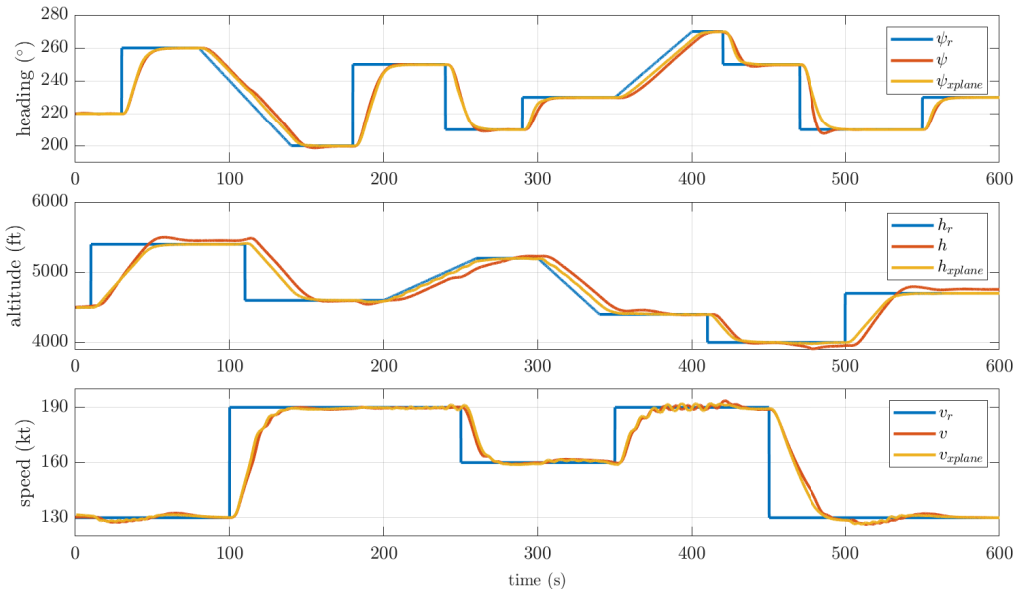


Fig. 7: Comparison of neural autopilot and X-Pane autopilot responses for the same reference signals

design tool was elaborated based on three modules. First, with the presentation of a co-simulation between X-Plane and Matlab, the paper gives the key features to help researchers benefit from this accurate and flexible flight simulator. This study then provides an offline black-box identification method based on neural networks that can suit to control theory framework. Finally, fault-tolerant control is proposed through an NMMAC. The approach relevance is shown by designing a speed sensor fault-tolerant flight director based on the data from the autopilot available in the simulator. This application demonstrates the proposed platform capabilities, which is promising for addressing more realistic cases in the future.

VII. The neural autopilot training platform: An efficient tool for robust autonomous controller deployment

This paper provides an illustration of the proposed method; however, the platform offers a much broader application scope than the one presented.

Indeed, this work aims to imitate X-Plane autopilot. However, in the future approach, the data will be generated using a human pilot to build an autopilot, reproducing its behaviour automatically. It is conceivable to use real flight data to develop the autopilot. It is also possible to shift the control law to another control stage than guidance, such as steering or even navigation. Finally, the work is not specific to aircraft since the multipurpose method using input-output data from a system and an existing controller to provides a nonlinear self-adaptive controller.

Due to the stability analysis challenge, the proposed autopilot will primarily assist the pilot. Nevertheless, in future works, the learning must be performed under stability constraint. The use of state-space neural networks opens up new analysis perspectives, particularly with the help of their already established links with Linear Fractional Transformation (LFT) and the associated robust control work [25], [26], [30]. The approach suggests the possibility of providing stability margins of the neural controller useful for certification.

Finally, the neural autopilot based on the NMMAC will become more and more intelligent and increase its reliability with the development of additional models based on new data. Its robustness will also be enhanced with observers considering an augmented state with disturbance.

References

[1] H. Baomar and P. J. Bentley, "An intelligent autopilot system that learns flight emergency procedures by imitating human pilots," in *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, 2016, pp. 1–9.

[2] M. Agha, K. Kanistras, P. C. Saka, K. Valavanis, and M. Rutherford, "System identification of circulation control uav using x-plane flight simulation software and flight data," in *AIAA Modeling and Simulation Technologies Conference*, 2017, p. 3154.

[3] C. W. Thong, "Modeling aircraft performance and stability on x-plane," *Australian Defence Force Academy, Univ. of New South Wales, Canberra, Australia*, 2010.

[4] A. Bittar, H. V. Figuereido, P. A. Guimaraes, and A. C. Mendes, "Guidance software-in-the-loop simulation using x-plane and simulink for uavs," in *2014 International Conference on Unmanned Aircraft Systems (ICUAS)*. IEEE, 2014, pp. 993–1002.

[5] L. R. Ribeiro and N. M. F. Oliveira, "Uav autopilot controllers test platform using matlab/simulink and x-plane," in *2010 IEEE Frontiers in Education Conference (FIE)*. IEEE, 2010, pp. S2H–1.

[6] R. Garcia and L. Barnes, "Multi-uav simulator utilizing x-plane," in *Selected papers from the 2nd International Symposium on UAVs, Reno, Nevada, USA June 8–10, 2009*. Springer, 2009, pp. 393–406.

[7] H. V. Figueiredo and O. Saotome, "Simulation platform for quadricopter: Using matlab/simulink and x-plane," in *2012 Brazilian Robotics Symposium and Latin American Robotics Symposium*. IEEE, 2012, pp. 51–55.

[8] A. Kaviyarasu and K. S. Kumar, "Simulation of flapping-wing unmanned aerial vehicle using x-plane and matlab/simulink," *Defence Science Journal*, vol. 64, no. 4, p. 327, 2014.

[9] E. Ersoy and M. K. Yalçin, "Designing autopilot system for fixed-wing flight mode of a tilt-rotor uav in a virtual environment: X-plane," *International Advanced Researches and Engineering Journal*, vol. 2, no. 1, pp. 33–42, 2018.

[10] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314, 1989.

[11] H. Baomar and P. J. Bentley, "Autonomous navigation and landing of airliners using artificial neural networks and learning by imitation," in *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*, 2017.

[12] K. Sezginer and C. Kasnakoğlu, "Autonomous navigation of an aircraft using a narx recurrent neural network," in *2019 11th International Conference on Electrical and Electronics Engineering (ELECO)*. IEEE, 2019, pp. 895–899.

[13] D. Shukla, S. Keshmiri, and N. Beckage, "Imitation learning for neural network autopilot in fixed-wing unmanned aerial systems," in *2020 International Conference on Unmanned Aircraft Systems (ICUAS)*. IEEE, 2020, pp. 1508–1517.

[14] M. Athans, D. Castanon, K.-P. Dunn, C. Greene, W. Lee, N. Sandell, and A. Willsky, "The stochastic control of the f-8c aircraft using a multiple model adaptive control (mmac) method–part i: Equilibrium flight," *IEEE Transactions on Automatic Control*, vol. 22, no. 5, pp. 768–780, 1977.

[15] G. J. J. Ducard, *Fault-tolerant flight control and guidance systems: practical methods for small unmanned aerial vehicles*, ser. Advances in industrial control. Springer, 2009.

[16] S. Fekri, D. Gu, I. Postlethwaite, and M. Athans, "Robust adaptive fault-tolerant control of the f-14 aircraft under sensor failures," *IFAC Proceedings Volumes*, vol. 41, no. 2, pp. 10 172–10 177, 2008.

[17] P. Bauer, R. Venkataraman, B. Vanek, P. J. Seiler, and J. Bokor, "Fault detection and basic in-flight reconfiguration of a small uav equipped with elevons," *IFAC-PapersOnLine*, vol. 51, no. 24, pp. 600–607, 2018.

[18] G. Aschauer, A. Schirrer, and M. Kozek, "Co-simulation of matlab and flightgear for identification and control of aircraft," *IFAC-PapersOnLine*, vol. 48, no. 1, pp. 67–72, 2015.

[19] M. T. Hagan, H. B. Demuth, M. H. Beale, and O. De Jess, *Neural network design*. Martin Hagan, 2014.

[20] M. Garzon and F. Botelho, "Dynamical approximation by recurrent neural networks," *Neurocomputing*, vol. 29, no. 1-3, pp. 25–46, 1999.

[21] L. Jin, M. M. Gupta, and P. N. Nikiforuk, "Dynamic recurrent neural networks for approximation of nonlinear systems," *IFAC Proceedings Volumes*, vol. 32, no. 2, pp. 5213–5218, 1999.

[22] P. M. Nørgård, O. Ravn, N. K. Poulsen, and L. K. Hansen, "Neural networks for modelling and control of dynamic systems-a practitioner's handbook," 2000.

[23] P. Gil, J. Henriques, A. Dourado, and H. Duarte-Ramos, "On state-space neural networks for systems identification: Stability and complexity," in *2006 IEEE Conference on Cybernetics and Intelligent Systems*. IEEE, 2006, pp. 1–5.

[24] N. Lachhab, H. Abbas, and H. Werner, "A neural-network based technique for modelling and lpv control of an arm-driven inverted pendulum," in *2008 47th IEEE Conference on Decision and Control*. IEEE, 2008, pp. 3860–3865.

[25] J. A. Suykens, J. P. Vandewalle, and B. L. de Moor, *Artificial neural networks for modelling and control of non-linear systems*. Springer Science & Business Media, 1995.

[26] H. Abbas and H. Werner, "Lpv design of charge control for an si engine based on lft neural state-space models," *IFAC Proceedings Volumes*, vol. 41, no. 2, pp. 7427–7432, 2008.

[27] S. Fekri, M. Athans, and A. Pascoal, "Issues, progress and new results in robust adaptive control," *International Journal of Adaptive Control and Signal Processing*, vol. 20, no. 10, pp. 519–579, 2006.

[28] F. Sims, D. Lainiotis, and D. Magill, "Recursive algorithm for the calculation of the adaptive Kalman filter weighting coefficients," *IEEE Transactions on Automatic Control*, vol. 14, no. 2, pp. 215–218, Apr. 1969.

[29] S. Fekri, "Robust adaptive mimo control using multiple-model hypothesis testing and mixed-μ synthesis," Ph.D. dissertation, Ph. D. dissertation, Instituto Superior Tecnico, Lisbon, Portugal, 2005.

[30] J. D. Bendtsen and K. Trangbæk, "Transformation of neural state space models into lft models for robust control design," in *Transformation of Neural State Space Models into LFT Models for Robust Control Design*, 2000.