Faculty of Aerospace Engineering
Systems Identification for Aerospace Vehicles
AE4320

Practical Assignment Number 2

Neural Networks

Responsible Professor: Dr. ir. C.C. de Visser

Authors and **Student Number**

Rohan Chotalal, 4746317

(Soft) Deadline: June 30, 2018
Delivered: March 27, 2019

2017/2018

Contents

List of Figures

List of Tables

1 Introduction

The role of system identification techniques from flight data has come to the necessity of finding mathematical models for aerospace systems (from small drones to big systems such an aircraft or a spacecraft). In aircraft control and simulation, these techniques can be applied on controlling the aircraft in extreme conditions, for instance, when a faulty effect occurs (where Fault Tolerant Flight Control, FTFC, techniques are applied); stall modelling, or even to simulate the previous responses in high fidelity simulators to mimic the aircraft behaviour and train the pilots for such undesired events.

These techniques rely on measurements obtained from the sensors as they are usually noisy and corrupted with a certain bias. Moreover, there are other states (x) of the system that cannot be measured directly. Thus, state estimation techniques should be implemented to make unbiased and noise-free predictions of the data. One of the well known techniques is called Kalman Filter (KF), where the system is composed by two models: a model that describes its own dynamics and another one which corresponds to dynamics of the sensors. Both of these models are assumed to be corrupted by artificial noise (Gaussian White Noise), which are the process and sensor noise, respectively, and whose knowledge is established by the user through the covariance matrices Q and R. The filter consists of two steps: a prediction step and a filtering/estimation step; its working principle is accounted with the calculation of a weighted average between the measured and predicted state: $x_{estimated} = x_{predicted} + K \cdot (z_{measured} - z_{predicted})$, where K is the Kalman gain.

There are many variants of this filter, which depends on the mathematical representation of the system. A Linear Kalman Filter is used if both the model and sensor dynamics are linear. However, if either one of them is non-linear, an Extended or Iterative Extended Kalman Filter (EKF and IEKF, respectively) can be used. The later are rater expensive computationally, as they require a linearisation of the system. Additionally, the IEKF takes a extra step in the filtering part to ensure an accurate estimation.

Parameter estimation techniques are useful to determine the parameters (θ) that best suit a pre-chosen model ($p(x, \theta)$) to fit the data-set. These rely on minimizing a certain cost function J which depends on the error ϵ between the ground-truth measurement (y) and the model output: $\epsilon = y - p(x, \theta)$. It corresponds to $\hat{\theta} = \text{argmin } (J(\epsilon))$. Note that the structure of the data as well as the complexity of the model defines how these parameters are estimated.

From the model point of view, the focus can be attributed to polynomial and non-polynomial models. The first attempts describe the input-output relation as a linear regression, and examples of such are simple polynomials or more advanced structures such as Multivariate Splines. Spline is a piecewise function with a predefined continuity between the pieces. In the multidimensional case, multivariate simplex splines are used. These are composed by an aglomeration of ordinary polynomial structures called simplices and whose geometric structure span a set of dimensions.

Non-polynomial parametric models are defined by a composition of linear and non-linear functions (here called neurons) whose parameters (known as weights) are not written in a linear regression form. A particular example of this kind functions is an Artificial Neural Network (ANN). This is a more advanced mathematical representation that mimics a biological neural network and is known as a "black-box" function approximator. It is constituted by a input layer, one (or more) hidden layers and one output layer. The interconnection between the layers is done by its neurons and the number of inputs and outputs is established by the input/output mapping, whereas the size (i.e. number of neurons) and number of hidden layers is chosen by the user. Optimization techniques that minimise a chosen cost function J are used to adjust the weights to the given mapping. Usually, the data-set is split into a training and testing/validation sets. The first is used to optimize the network whereas the second is considered as unseen data where the network is assesses its generalisation power.

Basis functions are mappings that when summed and scaled properly can approximate any other functions. For one hidden layer, these are described by the neurons in ANN as global approximators, which produce non-sparse solution systems. In multivariate splines, the simplices are local basis functions that join together in a process called triangulation. This ensures efficient sparse solution systems and avoids numerical stability inside the spline structure.

In the end, both ANN and splines are advanced system identification techniques that provide good approximation power and can fit any scattered nonlinear data set, which are the big advantages of these techniques. However, a huge drawback appears when the knowledge of the system is not known, making hard to understand its physics through the equations. Moreover, ANN requires a considerable amount data and training steps where a nonlinear optimization problem must be solved.

Structure of the report

In this report, the first part is directed towards the estimation of states from noisy data obtained from an aircraft, using a Kalman Filter. In the second part, the estimated states are used in a parameter estimation algorithm, named Ordinary Least Squares (OLS), with a linear parametric model. Furthermore, the same states

are also used to train two variants of non-linear models: Radial Basis Functions Neural Networks (RBF-NN) and Feed-Forward Neural Networks (FF-NN). The obtained results are compared with the ones obtained from the OLS approach but also validated for different choices of initial conditions and training setups.

2 State & Parameter estimation with F-16 flight data

In this section, two types of estimators will be introduced. The first one is a state estimator named Kalman Filter whereas the second one is a parameter estimator known as Ordinary Least Squares (OLS). Nevertheless, other parameter estimators will also be introduced.

Both methods and other variants will be used in the aerodynamic identification of the F-16 aircraft. In order to carry out simulations, data-set of the aircraft model is provided and its information is detailed in appendix A.

2.1 State Estimator - Kalman Filter

The Kalman Filter is a state estimation algorithm which uses series of measurements observed over time that contain statistical noise or other inaccuracies. It consists of two steps: prediction and filtering. Firstly, as the name states, the filter predicts the first estimation of the state. In the second step, the previous estimate is corrected with the Kalman gain, yielding the final estimation.

2.1.1 System state and observation models

A model of the system is usually described as a combination of two equations: the state equation and the observation equation, given in eq. (1), by $\mathbf{f}(\mathbf{x}, \mathbf{u})$ and $\mathbf{h}(\mathbf{x}, \mathbf{u})$, respectively. The state equation yields a mathematical description of the system model whereas the observation equation contains the same description but for the sensor model.

$$\begin{cases} \dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}) \\ \mathbf{z} = \mathbf{h}(\mathbf{x}, \mathbf{u}) \end{cases} \tag{1}$$

In this case, the state vector $\mathbf{x}$, input vector $\mathbf{u}$ and measurement vector $\mathbf{z}$ are represented by eqs. (2) to (4), respectively.

$$\mathbf{x} = \begin{bmatrix} u & v & w & C_{\alpha_{up}} \end{bmatrix}^T \tag{2}$$

$$\mathbf{z} = \begin{bmatrix} \alpha & \beta & V \end{bmatrix}^T \tag{3}$$

$$\mathbf{u} = \begin{bmatrix} \dot{u} & \dot{v} & \dot{w} \end{bmatrix}^T \tag{4}$$

The states (u, v, w) are the aircraft velocities and $(\dot{u}, \dot{v}, \dot{w})$ the accelerations, both obtained in the body frame. It is known that the measurements of the angle of attack (α) are affected by an unknown bias $(C_{\alpha_{up}})$. The other measurements, slide-slip angle (β) and true air speed (V), are assumed to perfect, i.e., not affected by any sensor inaccuracies.

To estimate the state vector $\mathbf{x}$, the body velocities can be obtained by integration of the body accelerations. Hence, the state equation is given by eq. (5). Note that the bias $C_{\alpha_{up}}$ is assumed to be constant, which means that its derivative is zero.

$$\mathbf{f}(\mathbf{x}, \mathbf{u}) = \begin{bmatrix} \dot{u} \\ \dot{v} \\ \dot{w} \\ 0 \end{bmatrix} \tag{5}$$

As for the observation (sensor) model, the following equations can be used:

$$\mathbf{h}(\mathbf{x}, \mathbf{u}) = \begin{bmatrix} \alpha_{true}(1 + C_{\alpha_{up}}) \\ \beta_{true} \\ V_{true} \end{bmatrix} = \begin{bmatrix} \arctan\left(\frac{w}{u}\right)(1 + C_{\alpha_{up}}) \\ \arctan\left(\frac{v}{\sqrt{u^2+w^2}}\right) \\ \sqrt{u^2 + v^2 + w^2} \end{bmatrix} \tag{6}$$

One can compute the α_{true} and β_{true} taking into account the geometry of the velocity vectors of the aircraft.

2.1.2 Kalman filter choice

The Kalman filter assumes that both state and sensor models are contaminated by a statistical noise, in this case, Gaussian White Noise (which follows a normal distribution with zero mean). Thus, adding a noise component in eq. (1) in both equations yields the extended model in eq. (7).

$$\begin{cases} \dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}) + G\mathbf{w} \\ \mathbf{z}_m = \mathbf{h}(\mathbf{x}, \mathbf{u}) + \mathbf{v} \end{cases} \tag{7}$$

where $\mathbf{w}$ and $\mathbf{v}$ are the process and sensor noise, respectively. Moreover, G is the noise matrix.

Thence, rewriting the state equation, it can be presented as in eq. (8), which is a linear state-space equation, because $\mathbf{f}(\mathbf{x}, \mathbf{u}) = F\mathbf{x} + B\mathbf{u}$. The state update depends directly in the input and noise.

$$\underbrace{\begin{bmatrix} \dot{u} \\ \dot{v} \\ \dot{w} \\ \dot{C_{\alpha_{up}}} \end{bmatrix}}_{\dot{\mathbf{x}}} = \underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}}_{F} \underbrace{\begin{bmatrix} u \\ v \\ w \\ C_{\alpha_{up}} \end{bmatrix}}_{\mathbf{x}} + \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}}_{B} \underbrace{\begin{bmatrix} \dot{u} \\ \dot{v} \\ \dot{w} \end{bmatrix}}_{\mathbf{u}} + \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{G} \underbrace{\begin{bmatrix} w_u \\ w_v \\ w_w \\ w_{C_{\alpha_{up}}} \end{bmatrix}}_{\mathbf{w}} \tag{8}$$

Following the same procedure with the observation model, eq. (9) is obtained.

$$\underbrace{\begin{bmatrix} \alpha_m \\ \beta_m \\ V_m \end{bmatrix}}_{\mathbf{z}_m} = \underbrace{\begin{bmatrix} \arctan\left(\frac{w}{u}\right)(1 + C_{\alpha_{up}}) \\ \arctan\left(\frac{v}{\sqrt{u^2+w^2}}\right) \\ \sqrt{u^2 + v^2 + w^2} \end{bmatrix}}_{\mathbf{h}(\mathbf{x},\mathbf{u})} + \underbrace{\begin{bmatrix} v_\alpha \\ v_\beta \\ v_V \end{bmatrix}}_{\mathbf{v}} \tag{9}$$

Looking at the expression, $\mathbf{h}(\mathbf{x}, \mathbf{u})$ is highly non-linear. Thus, it means that the linear Kalman Filter cannot be applied, because both state and sensor models should be linear. Thereafter, both the Extended and Iterative Extended Kalman Filters (EKF and IEKF, respectively) are the best choices as they contain a linearisation step on the system.

Comparing both EKF and IEKF, the first is not able to guarantee a global convergence to its optimal solution, as it depends on the initial condition. On the other hand, the IEKF is less prone to diverge, as it improves the convergence of the EKF through many filtering steps. However, its computation time is higher (even though this factor depends mainly in the computer's CPU - Central Processing Unit). In the present assignment, both filters were implemented and the results are compared.

2.1.3 Kalman filter implementation, results and convergence

Implementation

The implementation requires the definition of initial conditions to trigger the algorithm, which are the initial state ($\hat{x}_{0,0}$) and covariance matrix ($\hat{P}_{0,0}$). These are chosen by user and are the same for both EKF and IEKF.

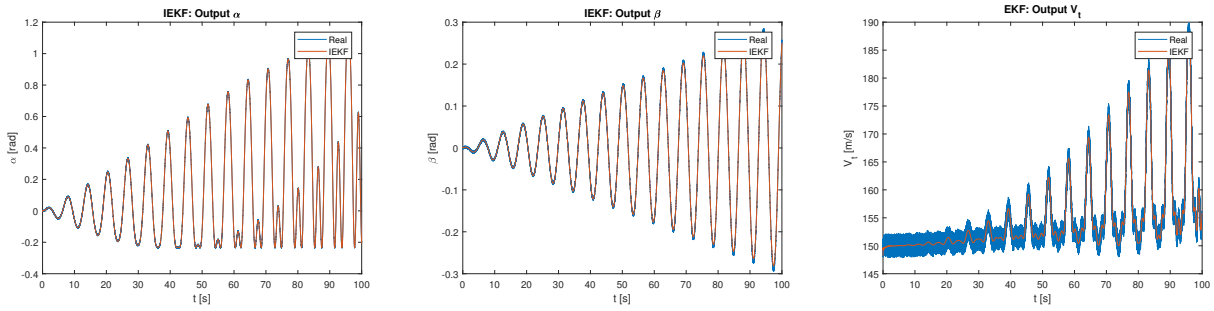
A choice for the initial state is presented in eq. (10). As mentioned earlier, the EKF is more sensitive to the initial condition when compared to the IEKF, because the algorithm is derived taking into account a first order approximation of the state and observation equations. Thus, the closer the initial estimate is to the optimal solution, more quickly it converges.

$$\hat{x}_{0,0} = \begin{bmatrix} 1 & 0 & 0 & 1 \end{bmatrix}^T \tag{10}$$

The trickiest choice is the covariance matrix. It contains information regarding the variance of the state estimation error. In other words, it establishes the confidence of the user in the chosen models. If high values are attributed to the entries of this matrix, the less trustful the model is, i.e., the initial estimates are less close to the optimal solution.

In this case, the matrix is square with size equal to number of states, which means 4, thus yielding a 4×4 matrix. This is chosen to be diagonal and with value 100 as shown in eq. (11).

$$\hat{P}_{0,0} = \begin{bmatrix} 100 & 0 & 0 & 0 \\ 0 & 100 & 0 & 0 \\ 0 & 0 & 100 & 0 \\ 0 & 0 & 0 & 100 \end{bmatrix} \tag{11}$$



(a) (b) (c)

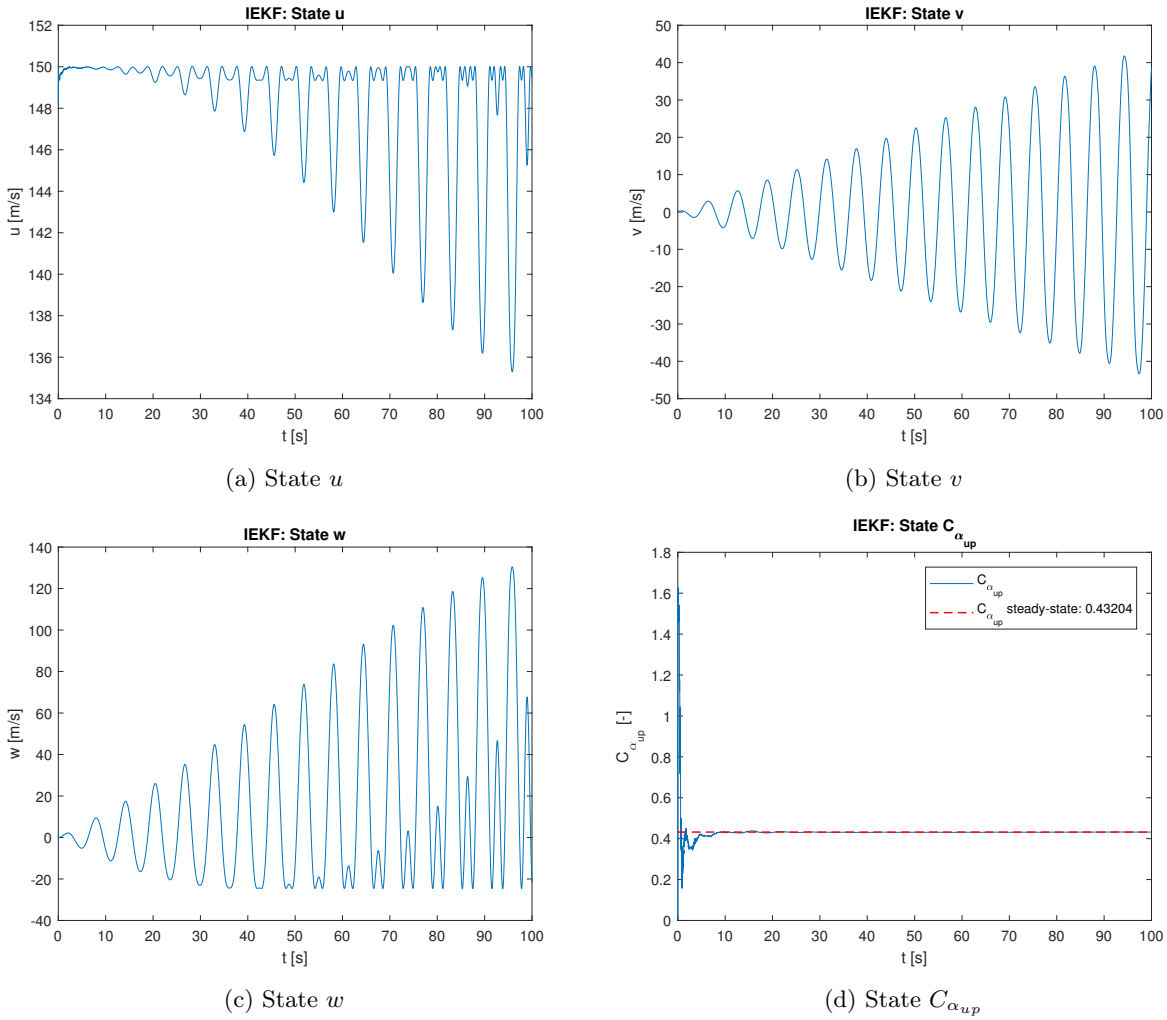
Figure 1: Ground-truth and IEKF estimation of the measurements: (a) α, (b) β and (c) V_t

Results

Since the IEKF gives better estimates in theory than the EKF, it will be used throughout the assignment. However, the results of the EKF (and comparison of both filters) is presented in appendix B.

Figure 1 shows the real measurements and the ones obtained from the IEKF which are computed through eq. (6) from the estimated states. It can be seen from the plots that the estimates are noise-free, has they present a smooth response, when compared with the real noisy measurements.

The estimated states are presented in fig. 2. Their real value is unknown, so there is no baseline to compare with. However, some conclusions can be taken, for instance, from bias $C_{\alpha_{up}}$ estimation: it varies a lot in the beginning and then remains almost constant through the rest of time. Thus, it means that the bias does not change and converges to $C_{\alpha_{up}} = 0.43204$.



(a) State u (b) State v

(c) State w (d) State $C_{\alpha_{up}}$

Figure 2: IEKF estimation of the states: (a) u, (b) v, (c) w and (d) $C_{\alpha_{up}}$

Convergence

There are many methods to evaluate the convergence of the Kalman Filter. However, in this report, two of them will be detailed:

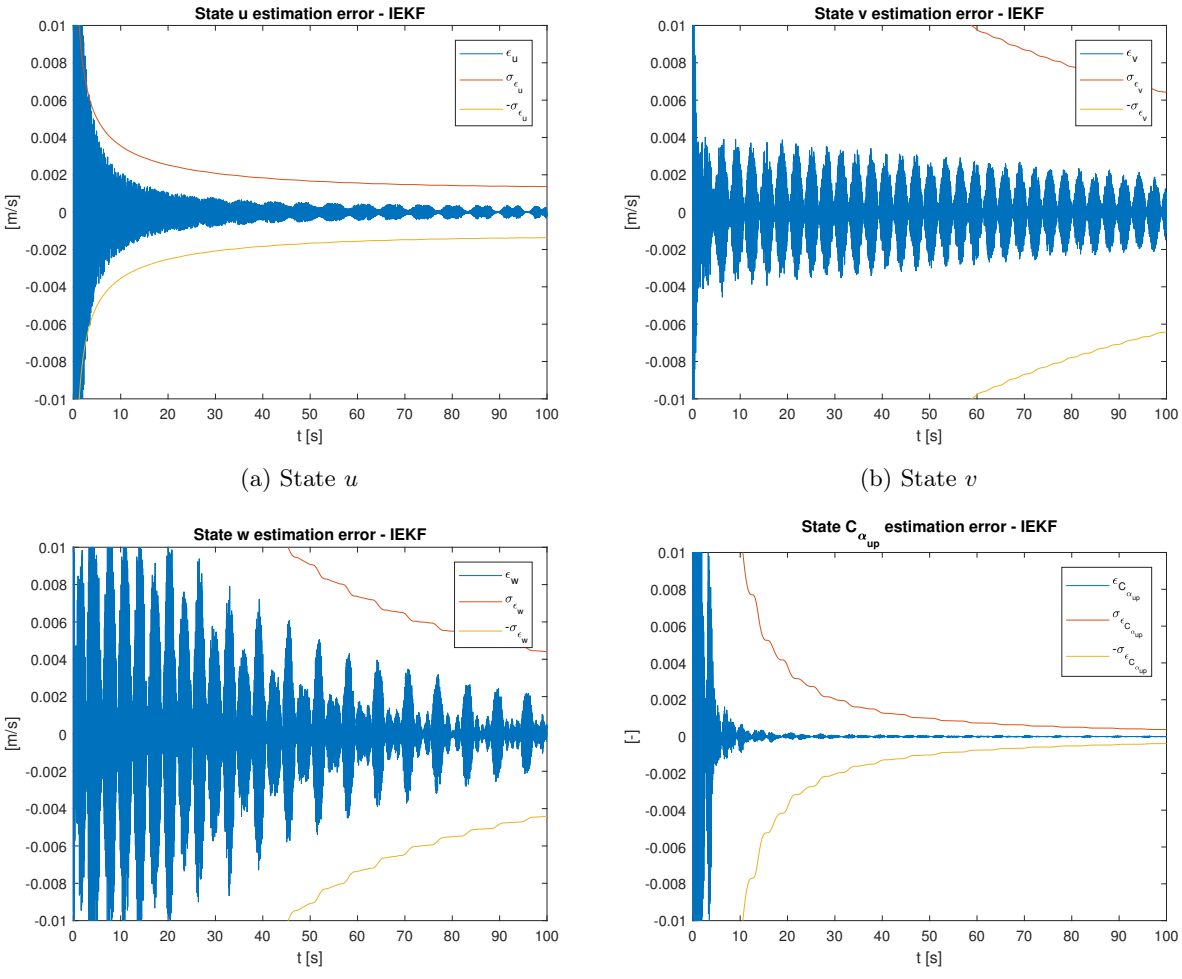
1. Analysis of the state estimation error

   This first method aims to be more experimental as it verifies if state estimation error tends to zero and is within the bounds of its standard deviation (obtained from the diagonal elements of state estimation error covariance matrix). The state estimation error is defined as the difference between the optimal state estimation ($\hat{x}_{k+1,k+1}$) and the real system state (x_{k+1}), as shown in eq. (12).

   $$\hat{\epsilon}_{k+1,k+1} = \hat{x}_{k+1,k+1} - x_{k+1} \tag{12}$$

   In this case, the actual state is unknown, which means that a second definition must be considered. This is an approximation of the difference between the optimal and predicted state estimates, as presented in eq. (13). Looking at the Kalman Filter algorithm, this difference is equal to filtering part of the algorithm, which is defined as a weighted average of the measured and predicted output, and usually responsible of the noise removal. But by using this definition, it is assumed high confidence on the predictive model, as close to the actual value of the states.

   $$\hat{\epsilon}_{k+1,k+1} \approx \hat{x}_{k+1,k+1} - \hat{x}_{k+1,k} \tag{13}$$



(a) State u

(b) State v

(c) State w

(d) State $C_{\alpha_{up}}$

Figure 3: IEKF state error estimation: (a) u, (b) v, (c) w and (d) $C_{\alpha_{up}}$

The results with this method are presented in fig. 3. As it can be seen, all the state estimation errors tend to zero, and stays within the bounds of the error standard deviation, which proves the convergence of the filter.

2. Observability analysis of the system

   This procedure is more analytical and takes into account the observability matrix (O) of the system (defined in eq. (14), because the system has 4 states). If its rank is equal to the number of states, the Kalman Filter converges, otherwise it diverges.

$$O = \begin{bmatrix} \partial_x h \\ \partial_x (L_f h) \\ \partial_x (L_f L_f h) \\ \partial_x (L_f L_f L_f h) \end{bmatrix} \tag{14}$$

   The implementation of this matrix is done with the help of the MatLab function `jacobian.m` in order to compute the derivative of the observation model with respect to the states, because $L_f h = \nabla h \cdot f$. This function is used inside a script developed by the responsible professor of this course and adapted for this particular project. Since the matrix's rank is equal to four for the initial conditions, the state is observable and consequently the Kalman Filter converges.

```
Rank of Initial Observability matrix is 3
    -> Rank of Observability matrix is 4
Observability matrix is of Full Rank: the state is Observable!
Augmented Observability matrix is of Full Rank: the augmented state is Observable!
```

   This approach is valid for EKF and IEKF as it is a more analytical than the previous one.

In the end, both methods confirm the convergence of the filter.

2.1.4 Reconstruction of the true angle of attack

After the implementation of the filters, it is possible to reconstruct the true value of α based on the estimated value of the bias $C_{\alpha_{up}}$. Thus, inverting eq. (15), eq. (16) is obtained:

$$\alpha = \alpha_{true}(1 + C_{\alpha_{up}}) \tag{15}$$

$$\alpha_{true} = \frac{\alpha}{1 + C_{\alpha_{up}}} \tag{16}$$

where $\alpha = \arctan(\frac{w}{u})$ is now determined using the estimated states from the Kalman Filter. This is an abuse of notation, because α_{true} was defined using the same definition. But now this variable is unknown as it is noise-free and unbiased.

A plot of both α_{true} and α_m (which is the original value) are shown in fig. 4 for the IEKF data set. α_{true} is a scaled version of α and α_m. This is already expected from the observation equation eq. (9) as it includes a noise and bias term.

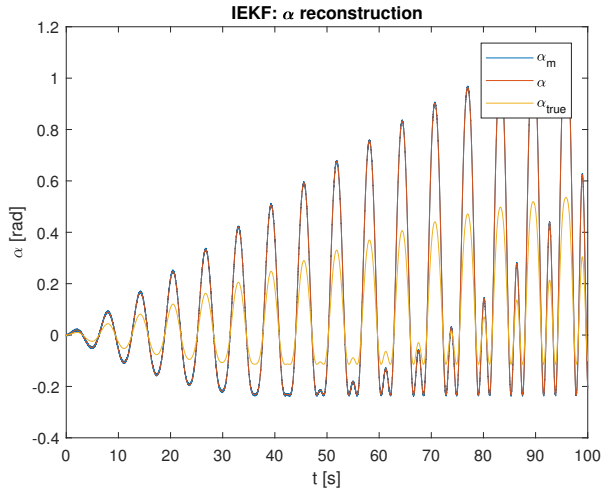


Figure 4: Reconstruction of the angle of attack for the IEKF

2.2 Parameter Estimator - Linear Least Squares algorithm

The Linear Least Squares (LLS) algorithm is a parameter estimator that gives a prediction of the weights or coefficients of a linear model, in this case, a polynomial model. These coefficients are calculated in such a way that it fits with dataset structure. There are many LLS algorithms such as Ordinary Least Squares (OLS), Weighted Least Squares (WLS), Generalized Least Squares (GLS) and Recursive Least Squares (RLS). Relevance is attributed to the OLS throughout the report.

2.2.1 Linear regression and Ordinary Least Squares (OLS) estimation for polynomial model

After removing the noise from the dataset with the help of the state estimator, the second step is to find a model that best fits the relation between α and β (which are an output of the Kalman Filter) with the pitching moment coefficient, C_m. The goal is to find a mathematical model that yields the following function $C_m = C_m(\alpha, \beta)$ for the given dataset.

Firstly, a model structure must be selected in order to apply the OLS algorithm. In this case, the multivariate polynomial model is chosen, which is an extension of the binomial theorem and defined ahead in eq. (17). This is written in a linear regression manner in eq. (18) which in matrix notation yields eq. (19):

$$C_m = C_m(\alpha,\beta) = \sum_{d=0}^{M} \sum_{n+m=d} C_{m_{\alpha^n\beta^m}} \frac{d!}{n!m!} \alpha^n \beta^m = \tag{17}$$

$$= \underbrace{C_{m_{\alpha^0\beta^0}}}_{C_{m_0}} + C_{m_{\alpha^1\beta^0}}\alpha^1\beta^0 + C_{\alpha^0\beta^1}\alpha^0\beta^1 + C_{m_{\alpha^2\beta^0}}\alpha^2\beta^0 + C_{m_{\alpha^1\beta^1}}\alpha^1\beta^1 + C_{m_{\alpha^0\beta^2}}\alpha^0\beta^2 + ... =$$

$$= C_{m_0} + C_{m_\alpha}\alpha + C_{m_\beta}\beta + C_{m_{\alpha^2}}\alpha^2 + C_{m_{\alpha,\beta}}\alpha\beta + C_{m_{\beta^2}}\beta^2 + ... \tag{18}$$

$$= \underbrace{\begin{bmatrix}1 & \alpha & \beta & \alpha^2 & \alpha\beta & \beta^2 & \dots\end{bmatrix}}_{a(\alpha,\beta)} \cdot \underbrace{\begin{bmatrix} C_{m_0} \\ C_{m_\alpha} \\ C_{m_\beta} \\ C_{m_{\alpha^2}} \\ C_{m_{\alpha\beta}} \\ C_{m_{\beta^2}} \\ \vdots \end{bmatrix}}_{\theta} = a(\alpha,\beta)\cdot\theta \tag{19}$$

where n and m are the exponents of α and β, respectively; M is the polynomial model order, $a(\alpha, \beta)$ is the regression vector and θ are the parameters to be estimated. The condition $n + m = d$ must always be verified, because the cross-terms $\alpha^n\beta^m$ (such as $\alpha\beta$, $\alpha^2\beta$, $\alpha\beta^2$, ...) are included, as they influence behaviour of the fitting. To apply the OLS algorithm, a regression matrix A is defined based on the N measurements ($\{(C_m^{(1)}, \alpha^{(1)}, \beta^{(1)})$, $(C_m^{(2)}, \alpha^{(2)}, \beta^{(2)}), ..., (C_m^{(N)}, \alpha^{(N)}, \beta^{(N)})\}$) in eq. (20).

$$\begin{bmatrix} C_m^{(1)} \\ C_m^{(2)} \\ \vdots \\ C_m^{(N)} \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & \alpha^{(1)} & \beta^{(1)} & (\alpha^{(1)})^2 & \alpha^{(1)}\beta^{(1)} & (\beta^{(1)})^2 & \dots \\ 1 & \alpha^{(2)} & \beta^{(2)} & (\alpha^{(2)})^2 & \alpha^{(2)}\beta^{(2)} & (\beta^{(2)})^2 & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \alpha^{(N)} & \beta^{(N)} & (\alpha^{(N)})^2 & \alpha^{(N)}\beta^{(N)} & (\beta^{(N)})^2 & \dots \end{bmatrix}}_{A=A(\alpha,\beta)} \cdot \underbrace{\begin{bmatrix} C_{m_0} \\ C_{m_\alpha} \\ C_{m_\beta} \\ C_{m_{\alpha^2}} \\ C_{m_{\alpha\beta}} \\ C_{m_{\beta^2}} \\ \vdots \end{bmatrix}}_{\theta} = \begin{bmatrix} a(\alpha^{(1)},\beta^{(1)}) \\ a(\alpha^{(2)},\beta^{(2)}) \\ \vdots \\ a(\alpha^{(N)},\beta^{(N)}) \end{bmatrix} \cdot \begin{bmatrix} C_{m_0} \\ C_{m_\alpha} \\ C_{m_\beta} \\ C_{m_{\alpha^2}} \\ C_{m_{\alpha\beta}} \\ C_{m_{\beta^2}} \\ \vdots \end{bmatrix} \tag{20}$$

Considering a second order model (i.e. $d = 2$), the results of the fitting are displayed in fig. 5 for the IEKF data. The black dots in plot represent the ground-truth values of C_m and the estimated coefficients of the model are presented in table 1 as an example.

C_{m_0}	C_{m_α}	C_{m_β}	$C_{m_{\alpha\beta}}$	$C_{m_{\alpha^2}}$	$C_{m_{\beta^2}}$
-0.06013	0.10665	-0.00094	-0.31778	-0.01565	0.17369

Table 1: OLS estimated coefficients of a 2nd order model for the IEKF data set

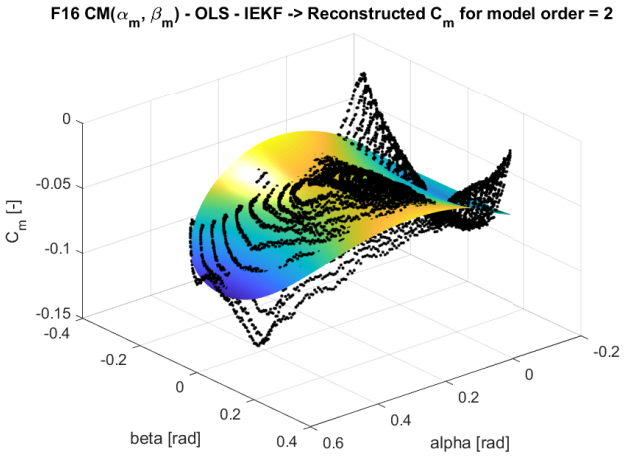


Figure 5: C_m reconstruction for IEKF data set using OLS algorithm with a second order polynomial

2.2.2 Model order influence

In order to assess the model error/residual, a variant of the Sum Squared Error (SSE) is used as a cost function $E = \frac{1}{2}\sum_{i=1}^{N}(C_m^{(i)} - a(\alpha^{(i)}, \beta^{(i)}) \cdot \theta)^2$, where $C_m^{(i)} - a(\alpha^{(i)}, \beta^{(i)}) \cdot \theta$ is the model residual for i-th measurement. This metric is used throughout the rest of report.

Therefore, a simulation was tested with the OLS from the 1rst to the 35th order of the polynomial model and is depicted in fig. 6. The bigger the order of the polynomial model, the more parameters it needs to be estimated, i.e, the model has more degrees of freedom.

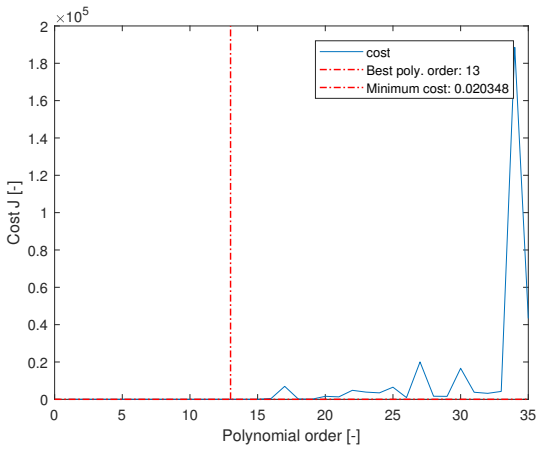


Figure 6: Evolution of the model performance with the polynomial order for the IEKF data set

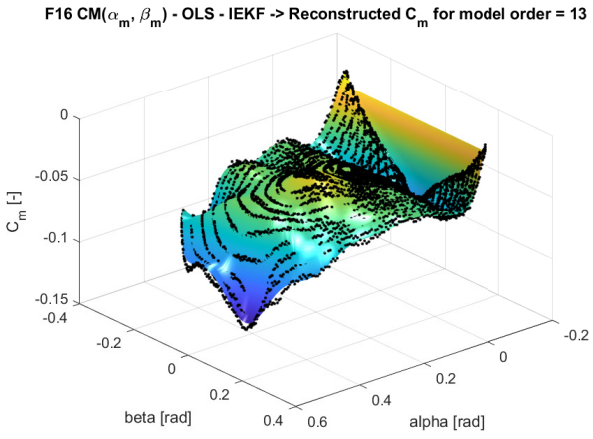


Figure 7: C_m reconstruction (with black dots as the ground-truth) for IEKF data set using OLS algorithm with a 13th order polynomial

It can be drawn from the plot that the cost decreases until the 13th order, where it achieves its minimum. The plot of this model is given in fig. 7. Afterwards, the algorithm overfits, i.e., it fails to generalize the data correctly which decreases its approximation power.

2.2.3 Model validation

Statistical-based validation

To perform a statistical-based model validation, one must check the variance of the estimated parameters. For the OLS algorithm these are obtained through the diagonal elements of the parameter covariance matrix which is given by $(A^TA)^{-1} = cov\{\hat{\theta}_{OLS}\} = E\{(\hat{\theta}_{OLS} - \theta)(\hat{\theta}_{OLS} - \theta)^T\}$. Computing this matrix for the best model order found in the previous section, the OLS algorithm yields the parameters in the bar graph shown fig. 8 and their respective variances in fig. 9.

The coefficients in the polynomial model that multiply with higher order terms tend to present higher variance. An explanation of this behaviour comes from the fact that the values of α and β are below one,

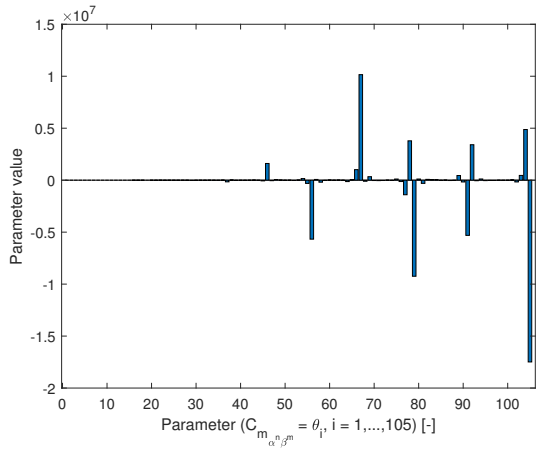


Figure 8: OLS estimated parameters of a 13th order model for the IEKF data set

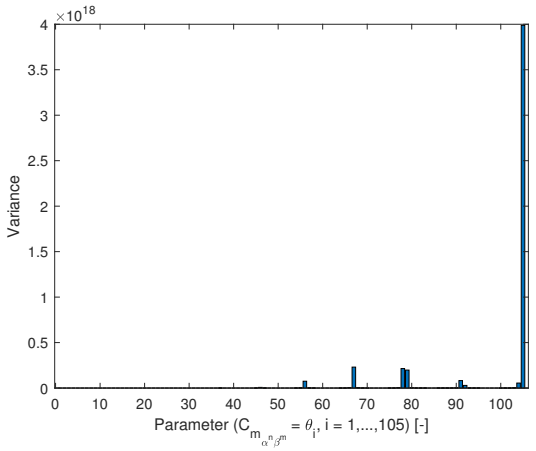


Figure 9: OLS parameter variances of a 13th order model for the IEKF data set

which means that the higher order terms of the linear regression become even smaller. On the other hand, the estimated parameters of this terms compensate them by presenting high absolute values. Thus, when the OLS algorithm computes the inverse of $A^T A$, the later is rater sensitive because there is a big difference in the absolute value of its entries.

The parameter covariances are the off-diagonal elements of this matrix. The entry with lowest value in this indicates the parameters that are correlated the most. For this case, are θ_{105} (last parameter of the linear regression) and θ_{78}, as the lowest value of this matrix is:

```
Parameters that are correlated the most: Cov{ theta_105, theta_78 } = -920457731436875776.000000
```

which corresponds to the entry that is in last row and 78th column.

Model-error validation

Instead of checking the parameters of the polynomial model, as seen the statistical-based validation, the model residual analysis takes into account the error of the model with respect to the ground truth value. The residual of the model is recalled as

$$\epsilon = C_m - a(\alpha, \beta) \cdot \theta \tag{21}$$

This validation analysis proceeds with the computation of the autocorrelation function (γ) of the residual, whose expression is given in eq. (22). This function measures the similarity between the observations (ϵ) as function of the time lag (l) between them (which is chosen by the user) and can be computed using the `xcorr` function in MatLab.

$$\gamma(l) = \sum_{i=-N}^{N} \epsilon(i)\epsilon(i+l) \tag{22}$$

This metric permits the user check the whiteness of the residual, i.e., whether the noise in C_m can be approximated as Artificial White Gaussian Noise (AWGN) process. The autocorrelation function of these type of noise processes is characterized by non-zero value for $l = 0$ and zero for rest of the lags.

In the end, implementing this function for the best model order, a plot is obtained in fig. 10 for $l = 300$. The 95% confidence bounds are determined by eq. (23) is used to verify that the for $l \neq 0$ the function lies within the interval $[-conf, +conf]$. Around $l = 0$, for a small number of lags, the function is outside the given boundaries. However, the AWGN approximation is still valid, because the function achieves a maximum value at that point, thus satisfying the property characterised earlier.

$$conf = \frac{1.96}{\sqrt{N}} \tag{23}$$

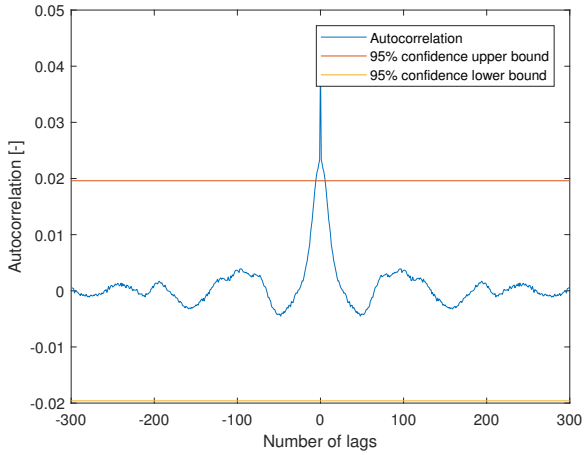


Figure 10: Autocorrelation function of the 13th order model residual for the IEKF data set

3 Radial Basis Function Neural Network (RBF-NN)

Neural networks are advanced non-linear structures used to give a relationship between the input and output of a given dataset. In this assignment, the goal of these networks is similar to the OLS algorithm: the reconstructed data from the Kalman Filter is used create a continuous mapping of α and β with C_m. The parameters of these models (here called 'weights') are adjusted during learning procedure by means of optimization techniques.

In this section, Radial Basis Function Neural Network (RBF-NN) are used (fig. 11). It contains three layers: one input layer with two inputs $u = (u_1, u_2)$ (where $u_1 = \alpha$ and $u_2 = \beta$), one hidden layer with n RBFs (neurons of the network) whose number is selected by the user, and an output layer with single output ($y = \hat{C}_m \approx C_m$). The j-th RBF activation function is presented in eq. (25) and follows a normal distribution whose location, width and amplitude are described, respectively, by the centers c_{1j} and c_{2j}, the input/inner weights (w_{1j} and w_{2j}) and the output/outer weights w_j (denoted as a in the assignment).

$$y = \sum_{j=1}^{n} w_j \phi_j = \hat{C}_m \tag{24}$$

$$\phi_j = \exp\{-v_j\} \tag{25}$$

$$v_j = \sum_{i=1}^{2} w_{ij}^2 (u_i - c_{ij}) = w_{1j}^2 (u_1 - c_{1j})^2 + w_{2j}^2 (u_2 - c_{2j})^2 \tag{26}$$

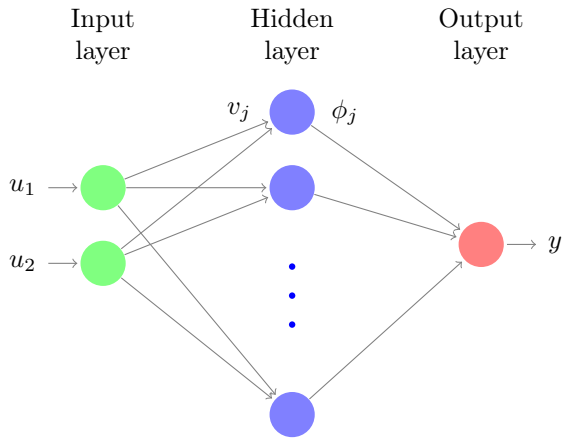


Figure 11: Representation of a RBF-NN

3.1 Reconstructed data set approximation using linear regression to RBF-NN

The output of the RBF-NN is obtained through a weighed sum of all the outputs of the neurons (in eq. (24)). To estimate these weights, a similar approach to eq. (18) is presented in eq. (27), writing the outputs of the neurons in a linear regression manner, where b is the regression vector and θ is the vector of parameters obtained through OLS algorithm.

$$y = \sum_{j=1}^{n} w_j \phi_j = w_1\phi_1 + w_2\phi_2 + ... + w_n\phi_n = \underbrace{\begin{bmatrix}\phi_1 & \phi_2 & ... & \phi_n\end{bmatrix}}_{b} \underbrace{\begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}}_{\theta} = b \cdot \theta \tag{27}$$

Comparing to eq. (20), the model can be written for N measurements, which yields the regression matrix B in eq. (28). Note that $\phi_j^{(q)} = \exp\{-v_j^{(q)}\} = \exp\{-[w_{1j}^2(u_1^{(q)} - c_{1j})^2 + w_{2j}^2(u_2^{(q)} - c_{2j})^2]\}$, where superscript indicates the q-th data-point/measurement: $\{u_1^{(q)} = \alpha^{(q)}, u_2^{(q)} = \beta^{(q)}, y = C_m^{(q)}\}$.

$$\begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(N)} \end{bmatrix} = \underbrace{\begin{bmatrix} \phi_1^{(1)} & \phi_2^{(1)} & \dots & \phi_n^{(1)} \\ \phi_1^{(2)} & \phi_2^{(2)} & \dots & \phi_n^{(2)} \\ \phi_1^{(3)} & \phi_2^{(3)} & \dots & \phi_n^{(3)} \\ \vdots & \vdots & \vdots & \vdots \\ \phi_1^{(N)} & \phi_2^{(N)} & \dots & \phi_n^{(N)} \end{bmatrix}}_{B} \cdot \underbrace{\begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}}_{\theta} = \begin{bmatrix} b^{(1)} \\ b^{(2)} \\ b^{(3)} \\ \vdots \\ b^{(N)} \end{bmatrix} \cdot \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} \tag{28}$$

The inner weights w_{ij} and centers c_{ij} are defined by the user. In order for the network to converge, the centers of the RBF are initialised using a k-means clustering algorithm ([4, 5]). This is done using the pre-built MatLab function called `kmeans.m`. It depends on the number of neurons chosen to segment the input data into $k = n$ clusters and, thus, find the centroid of each one. These will be the locations of the RBFs.

For the inner weights, the expression of an RBF neuron can be compared with the probability density function (pdf) of a bivariate normal distribution (eq. (29)) of two uncorrelated ($cov\{u_1, u_2\} = 0$) random variables u_1 and u_2. Note that (μ_1, μ_2) corresponds to the mean of the distribution.

$$p(u_1, u_2) = \frac{1}{2\pi\sigma_{U_1}\sigma_{U_2}} \exp\left\{-\frac{1}{2}\left[\frac{(u_1 - \mu_{u_1})^2}{\sigma_{U_1}^2} + \frac{(u_2 - \mu_{u_2})^2}{\sigma_{U_2}^2}\right]\right\} \tag{29}$$

The standard deviations σ_{U_i} (for $i \in \{1, 2\}$) define the width of the pdf in the u_i direction. The bigger its value the wider the radius of the distribution in that direction. Therefore, an inverse relationship of the RBF radius and its inner weight in the i-th direction can be obtained in eq. (30) by comparing the argument of the exponential function in eq. (29) with eq. (26). If the inner weight is higher, the radius is lower. If the inner weight presents a low value, the radius of the RBF is higher.

$$\frac{1}{2\sigma_i^2} = w_{ij}^2 \iff \sigma_i = \sqrt{\frac{1}{2w_{ij}^2}} \tag{30}$$

Furthermore, looking at the expression, these weights should be initialised with positive values, otherwise the argument of the exponential diverges when a value smaller than -1 is attributed. In this case, these values randomly drawn from an uniform distribution between $[0, x]$, where x in this case is 2. To study the sensitivity of these weights in the network's performance, an algorithm was developed and explained in appendix appendix C.

Thus, considering a network with 5, 10, 15 and 20 neurons, the reconstruction of the C_m for the IEKF data set is obtained alongside with the position of RBFs. Results for a network with 5 are shown in figs. 12 and 13, and for 20 neurons are shown figs. 14 and 15. The radius of circles shown in figs. 13 and 15 does not correspond to the actual radius of the RBFs. It is just a mere representation of where they are located.

The RBFs are placed within the input boundaries of data set using the k-means algorithm. Choosing the same variant of the SSE defined in 2.2.2 as a performance metric, different results are presented in table 2. The cost function decreases with an increase of the number of neurons. However, this is not a linear relationship, and can be seen in section 3.3 that the model overfits from a certain number of neurons. For instance, from 15 to 20 neurons, the performance slightly decreases.

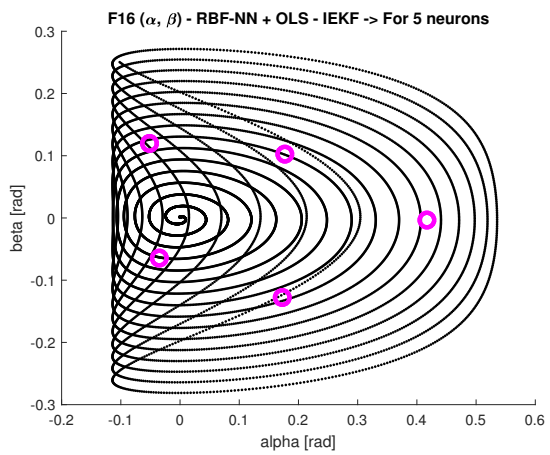


Figure 12: Position of the centers using the k-means clustering algorithm for a RBF-NN with 5 neurons (IEKF data set)

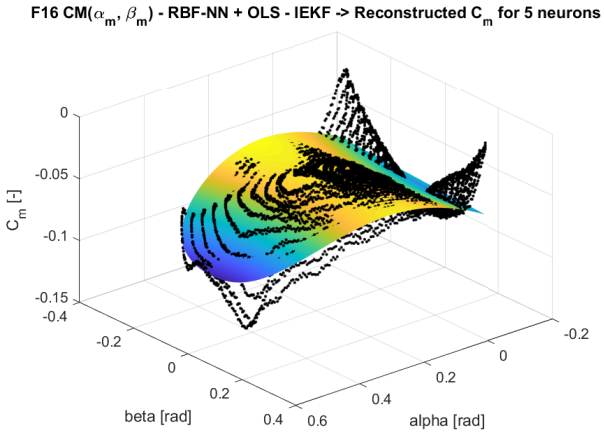


Figure 13: C_m reconstruction (with black dots as the ground-truth) obtained for a RBF-NN with 5 neurons using the OLS algorithm (IEKF data set)

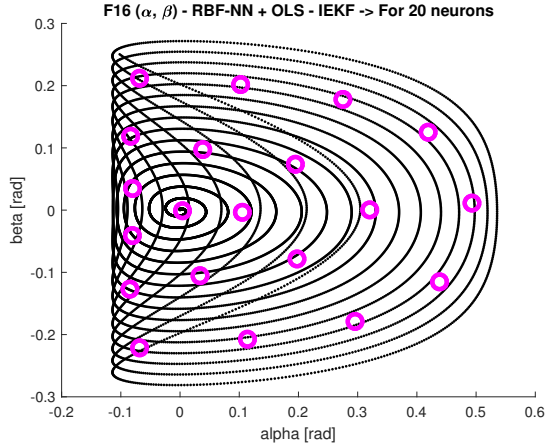


Figure 14: Position of the centers using the k-means clustering algorithm for a RBF-NN with 20 neurons (IEKF data set)

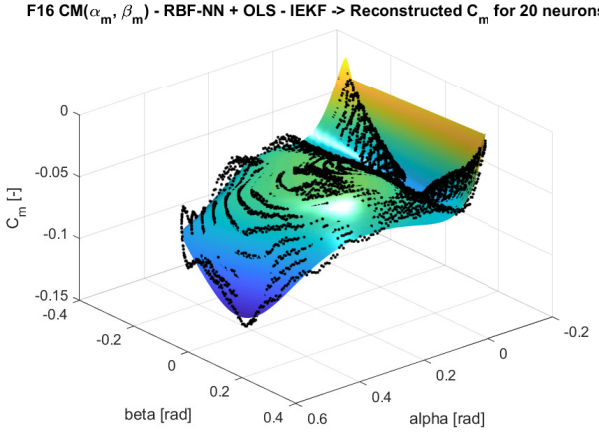


Figure 15: C_m reconstruction (with black dots as the ground-truth) obtained for a RBF-NN with 20 neurons using the OLS algorithm (IEKF data set)

Number of neurons [–]	Cost J [–]
5	0.644242
10	0.329964
15	0.137169
20	0.138707

Table 2: Performance of the RBF-NN trained with the OLS algorithm for different choice of neurons (IEKF data set)

3.2 RBF-NN learning using the Levenberg-Madquardt (LM) algorithm

A neural network can be learned (i.e., update the weights) by means of optimization algorithms. One of the most common optimizers is Levenberg-Madquardt (LM) algorithm. Generally, the network is trained every epoch k so that the weights are updated through eq. (31).

$$\omega_{k+1} = \omega_k + \Delta\omega_k \tag{31}$$

The goal of the optimizer is to minimise the same cost function defined in section 2.2 and here recalled in eq. (32) as a variant of the sum squared errors of all the data points. Note that the network is trained in a supervised manner, which means that there is a ground-truth label C_m (here called 'target' d) for each input $\{\alpha^{(q)}, \beta^{(q)}\}$. $y = \hat{C}_m$ are the predicted outputs of the network.

$$E = \frac{1}{2}\sum_{q=1}^{N}(e^{(q)})^2 \tag{32}$$

$$e = d - y = C_m - \hat{C_m} \tag{33}$$

For the LM algorithm, the weight updates are given by eq. (34). It is expressed as a combination of two methods ([1]): 2nd order Gauss-Newton optimizer ($-(J^TJ)^{-1}J^Te$) and, a 1rst order Gradient Steepest Descent algorithm ($-\lambda^{-1}J^Te$) used in section 4.1 for the Feedfoward Neural Network. J is the Jacobian matrix and can be defined based on the learning type: online learning (generally known as stochastic learning) or batch learning. The two learning methods are explained in [2].

$$\Delta\omega_k = -(J^TJ + \lambda I)^{-1}J^Te \tag{34}$$

In this report, the training takes into account the entire data set as a batch. Since each RBF neuron has 5 weights, for n neurons there are $5n$ weights to learn. Therefore, matrix J is defined in eq. (35), which contains the derivatives of the network error with respect to these weights (in each column) and for all the data points (in each row) ([6]).

$$J = \begin{bmatrix} \frac{\partial e^{(1)}}{\partial w_{11}} & \cdots & \frac{\partial e^{(1)}}{\partial w_{1n}} & \frac{\partial e^{(1)}}{\partial w_{21}} & \cdots & \frac{\partial e^{(1)}}{\partial w_{2n}} & \frac{\partial e^{(1)}}{\partial c_{11}} & \cdots & \frac{\partial e^{(1)}}{\partial c_{1n}} & \frac{\partial e^{(1)}}{\partial c_{21}} & \cdots & \frac{\partial e^{(1)}}{\partial c_{2n}} & \frac{\partial e^{(1)}}{\partial w_1} & \cdots & \frac{\partial e^{(1)}}{\partial w_n} \\ \frac{\partial e^{(1)}}{\partial w_{11}} & \cdots & \frac{\partial e^{(1)}}{\partial w_{1n}} & \frac{\partial e^{(1)}}{\partial w_{21}} & \cdots & \frac{\partial e^{(1)}}{\partial w_{2n}} & \frac{\partial e^{(1)}}{\partial c_{11}} & \cdots & \frac{\partial e^{(1)}}{\partial c_{1n}} & \frac{\partial e^{(1)}}{\partial c_{21}} & \cdots & \frac{\partial e^{(1)}}{\partial c_{2n}} & \frac{\partial e^{(1)}}{\partial w_1} & \cdots & \frac{\partial e^{(1)}}{\partial w_n} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{\partial e^{(N)}}{\partial w_{11}} & \cdots & \frac{\partial e^{(N)}}{\partial w_{1n}} & \frac{\partial e^{(N)}}{\partial w_{21}} & \cdots & \frac{\partial e^{(N)}}{\partial w_{2n}} & \frac{\partial e^{(N)}}{\partial c_{11}} & \cdots & \frac{\partial e^{(N)}}{\partial c_{1n}} & \frac{\partial e^{(N)}}{\partial c_{21}} & \cdots & \frac{\partial e^{(N)}}{\partial c_{2n}} & \frac{\partial e^{(N)}}{\partial w_1} & \cdots & \frac{\partial e^{(N)}}{\partial w_n} \end{bmatrix} \tag{35}$$

These derivatives are determined based on the chain rule method. For the RBF network they are computed from eqs. (36) to (38).

$$\frac{\partial e}{\partial w_j} = \frac{\partial e}{\partial y}\frac{\partial y}{\partial w_j} = (-1)\cdot\phi_j \tag{36}$$

$$\frac{\partial e}{\partial w_{ij}} = \frac{\partial e}{\partial y}\frac{\partial y}{\partial \phi_j}\frac{\partial \phi_j}{\partial v_j}\frac{\partial v_j}{\partial w_{ij}} = (-1)\cdot w_j\cdot(\exp\{-v_j\})\cdot(2w_{ij}(u_i - c_{ij})^2) \tag{37}$$

$$\frac{\partial e}{\partial c_{ij}} = \frac{\partial e}{\partial y}\frac{\partial y}{\partial \phi_j}\frac{\partial \phi_j}{\partial v_j}\frac{\partial v_j}{\partial w_{ij}} = (-1)\cdot w_j\cdot(\exp\{-v_j\})\cdot(-2w_{ij}^2(u_i - c_{ij})) \tag{38}$$

The parameter λ is a damping factor that weights the updates obtained from the steepest descent algorithm, it is usually called 'learning rate', and depends on the learning algorithm ([1]):

- Fixed learning, where this learning rate is constant the entire training process;
- Adaptive learning, where the learning rate varies during training based on an assessment of the previous and current cost function.

Both algorithms are detailed in appendix D. In the LM algorithm, for the adaptive case, if the cost at the current epoch decreases, the changes of the network are accepted and this factor is multiplied by γ^{-1}; however, if the cost increases, the changes are discarded, keeping the previous values of the weights and the learning rate is multiplied by γ (where $\gamma > 1$). For the fixed algorithm, the updates are only considered if the current cost decreases compared to the one in the previous epoch, otherwise the updated network continues to search for the best performance.

During the learning procedure, the data set is divided into 3 sets:

- Training set - used to carry the updates of the weights;
- Validation set - used to assess the network after each epoch;
- Testing set - used to assess the network approximation power after the training procedure.

Throughout this report, the testing set is used in conjunction with the validation set to also assess the network after each epoch. Note that both validation and testing sets are unseen data for the network, i.e., the network 'did not study/learn the behaviour of this data'. Usually a good division includes a high amount of

training data and rest for validation/testing. The following choice is considered hereafter (unless mentioned otherwise): 80%/10%/10% (training/validation/testing). This division will be relevant for the neural network structure optimization in the next subsection and in section 4.3.

A RBF-NN with 20 neurons is considered. Their centers are again initialized using the k-means clustering algorithm, the inner and outer weights are drawn randomly from a uniform distribution in the interval $[0, 2]$ and $[-2, 2]$, respectively. The network is trained for 3000 and 9000 epochs and the results of the C_m reconstruction, localization of the RBF centers after training and behaviour of the cost function for 9000 epochs are given from figs. 16 to 20. Note that the location of the centers before training was shown in previously in fig. 14. Moreover, the training, validation and testing input data points are also distinguished in figures figs. 16 and 18. The learning algorithm is adaptive (this is a standard choice thoughout the rest of the report), as the parameter λ is initialized with a value of 100 and varies at each epoch with $\gamma = 10$.

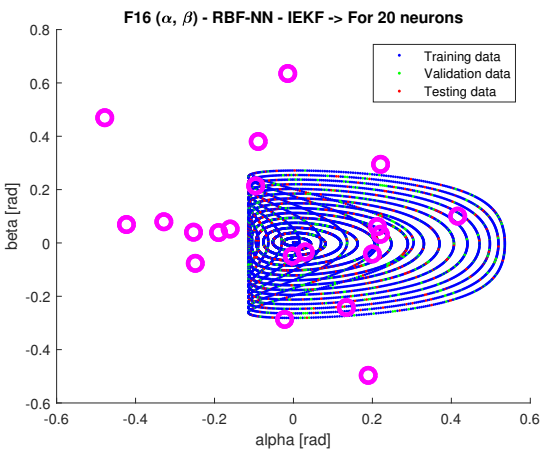


Figure 16: Position of the centers for a RBF-NN with 20 neurons after LM training for 3000 epochs (IEKF data set)

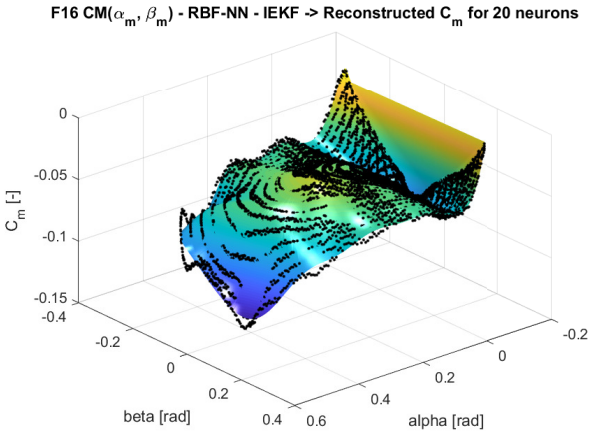


Figure 17: C_m reconstruction (with black dots as the ground-truth) obtained for a RBF-NN with 20 neurons after LM training for 3000 epochs (IEKF data set)

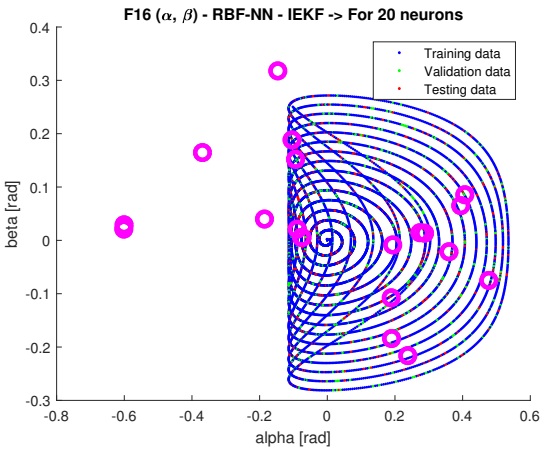


Figure 18: Position of the centers for a RBF-NN with 20 neurons after LM training for 9000 epochs (IEKF data set)

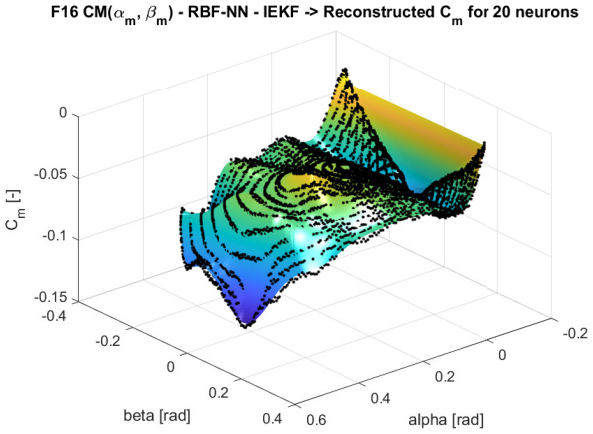


Figure 19: C_m reconstruction (with black dots as the ground-truth) obtained for a RBF-NN with 20 neurons after LM training for 9000 epochs (IEKF data set)

The optimization of the network's weights minimizes the previously mentioned cost function, i.e, the sum squared errors between the network predictions and the ground-truth predictions globally. However, the RBFs have a local behaviour and they can provide better results if the they are placed within the boundaries of the data set. The global optimization does not take into account this factor in the beginning, as it depends mainly on the input data structure, i.e, the placement of α and β. For instance, if the RBFs are optimally placed with the use of the k-means algorithm (or even random initialization within the boundaries of the data set) the training algorithm would update the locations outside of this area with the goal of decreasing the error between the target and predicted output.

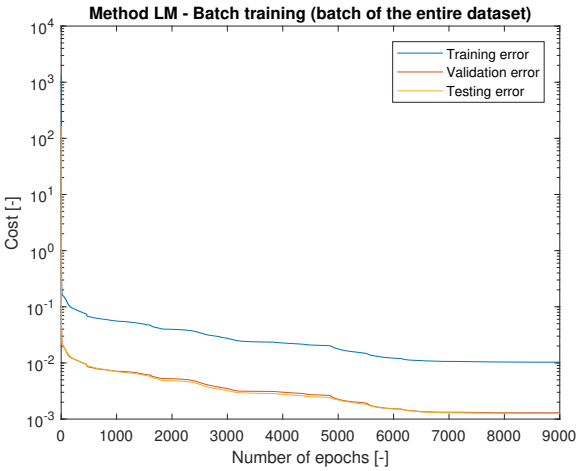


Figure 20: Cost function behaviour of a RBF-NN with 20 neurons after LM training for 9000 (IEKF data set)

Usually the goal is to train a network with the least number of iterations possible and obtain the best performance. Table 3 shows the performance of the network for different epochs. The training cost is expected to always decrease, as the network learns from this data. The validation and testing costs (E_{val} and E_{test}, respectively) also decrease. Thus, the total cost (E_{tot}), which is given by the sum of the previous costs, also decreases, and by 9000 epochs of training, the performance is better than the one obtained from the OLS algorithm in section 3.1. Visually, the reconstruction of C_m is also better for this network than the one obtained for 3000 epochs.

In the end, increasing the number of epochs induces a better approximation of the data set mapping. However, the best network is obtained when the validation/testing cost achieves a minimum value before it starts to increase. From that point on-wards, it starts to overfit. In this assignment, the data points are very close to each other, therefore it requires lots of epochs (and therefore, training time) to observe this behaviour. The major disadvantage compared to OLS algorithm is the time it takes to learn the data set structure.

Epoch [−]	E_{train} [−]	E_{val} [−]	E_{test} [−]	E_{tot} [−]
3000	0.0274	0.0035	0.0032	0.0341
6000	0.0122	0.0015	0.0015	0.0152
9000	0.0103	0.0013	0.0013	0.0129

Table 3: Performance of the RBF-NN with 20 neurons trained with the LM algorithm for different epochs (IEKF data set)

In order to evaluate the sensitivity of the network to the initial conditions, different values of initial damping factor varying from 10 to 1000 (in factors of 10) are used to train the network for 3000 epochs. This choice permits the observation of the final performance when trained for a small amount of time. The results are presented in table 4. From the three values shown in the table, the $\lambda = 100$ showed a good final performance. Fixing this initial value of λ, the network is trained for different choice of number of neurons for the same number of epochs and initial conditions. As the network structure increases, there are more weights to update, the performance also increases (except for the case of 15 neurons). However, only 3000 iterations are used in the learning procedure, which is not enough to surpass the performance obtained from the OLS algorithm.

λ [−]	E_{tot} [−]
10	0.0351
100	0.0341
1000	0.0367

Table 4: Performance of the RBF-NN with 20 neurons trained with the LM algorithm for different values of initial damping factor λ (IEKF data set)

Number of neurons [−]	E_{tot} [−]
5	0.0775
10	0.0370
15	0.0521
20	0.0341

Table 5: Performance of the RBF-NN trained with the LM algorithm for different choice of neurons (IEKF data set)

3.3 Number of neurons optimization for the RBF-NN

It is convenient to determine a model order that ensures good performance, i.e., the best input-output mapping of the data set through a complex model. Therefore, this section consists on the design of an optimization algorithm that indicates the best model structure of the RBF-NN in terms of number of neurons for both the OLS and LM algorithms and compare them.

For a given model order, the k-fold cross-validation algorithm is applied, where the data set is split in equal sized k folds/subsets. From the k subsets, one is split (equally) for testing and validation. The other $k-1$ subsets are merged to be used as a training set. The process is repeated k times so that every subset is used at least once as a test set and the results of the obtained cost are averaged for each one of the sets. On top of that, for each repetition, several weight initializations are considered, as the algorithm does not get stuck in a local minimum when averaging the results. In summary, for each fold the results are averaged once for different initializations and then a second average is considered for the results obtained from all the folds.

The number of folds is determined based on the fraction of training data, by means of the expression in eq. (39), where $\lfloor x \rfloor$ corresponds to the `floor` operator, where a real number x is approximated to the greatest integer less than equal to it.

$$k = \left\lfloor \frac{1}{1 - \text{fraction of training data}} \right\rfloor \tag{39}$$

In this case, 80% of the data set is used for training, which means that there are $k = \frac{1}{1-0.8} = \frac{1}{0.2} = 5$ folds: 4 used for training and the remnant used for testing/validation. Moreover, for each fold, 5 different random initializations are considered during training.

The neural network is trained considering a range of number of neurons from 1 to a maximum number defined by the user (here chosen as 35 neurons) and is validated by unseen data such as the testing set. The best structure is determined by the number of neurons that yields the smallest testing cost. It expected that the network overfits when the test set starts to present an increasing trend.

Hereby, two models of RBF-NN are trained: one using the OLS algorithm (from section 3.1) and another one using the LM algorithm. For LM algorithm, the weights are initialised in the same manner as described in the previous section. The initial value of λ is 100 and the maximum number of epochs is 400. The reasons of this choice for the number of epochs are:

- the algorithm is already computationally heavy which takes longer to train a network for models with more neurons. The learning algorithm must be run at least (5 folds) $\times$ (5 initializations) $= 25$ times for each choice of number of neurons. Moreover, an user would like to achieve the best possible model through less learning iterations;
- the cost of the RBF-NN starts to stabilise (although in a decreasing trend) with at least 400 epochs of training. In this case, the reconstruction of the pitching moment, C_m, is already acceptable (visually) when compared to an untrained network.

The results are presented in figs. 21 and 22.

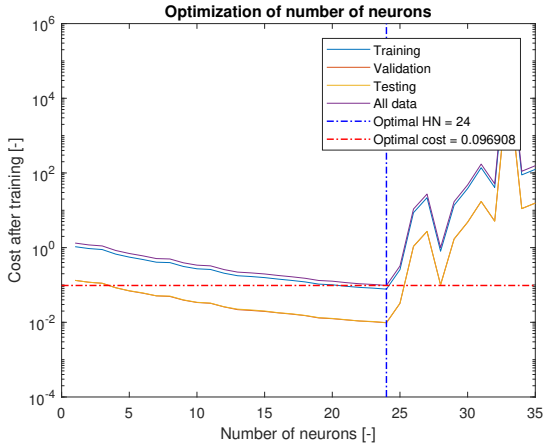


Figure 21: Model performance of the RBF-NN with the number of neurons using the OLS algorithm (IEKF data set)

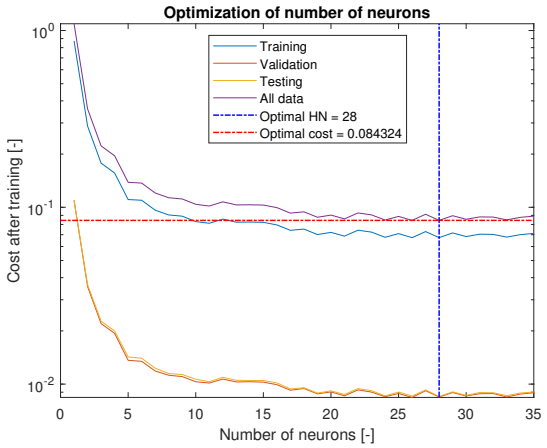


Figure 22: Model performance of the RBF-NN with the number of neurons using the LM algorithm (IEKF data set)

For the OLS algorithm, the optimal number of neurons that guarantees the best performance is 24. The total cost of a network with this structure is given (in the plot): 0.096908. From this point the model diverges,

because the approximation power of the network decreases as the cost starts to increase with the number of neurons. The algorithm only updates the amplitude of the RBFs. Thus, when the number of neurons increases, the radius of the RBF covers certain clusters of data, which can yield a good approximation of the data inside it but a bad approximation in the adjacent clusters. To avoid this problem, it is proposed to increase the value of the inner weights when the number of neurons increases. This will reduce the width/radius of the RBFs and therefore assure a more clever approximation.

For the LM algorithm, the optimum number of neurons is 28, whose total cost function after training is 0.084324. This value is smaller than the one obtained from the OLS algorithm, which shows a higher performance. The plot shows that all the sets (training/validation/testing) show a similar decreasing trend. The reason why this occurs is that the data points are very close to each other which makes it difficult for the model to overfit. By learning one data-point, the network also learns a small neighbourhood around it. There are 10001 data points, where 80%, which means, at least 8000 samples, are used for training whereas the rest is used for validation/testing. This explains the small cost in the later sets.

4 Feed-Forward Neural Network (FF-NN)

A Feed-Forward Neural Network (FF-NN) (fig. 23) shows a different behaviour than the RBF-NN. Each neuron's activation function is given by a tangent sigmoidal function eq. (41) whose input is a weighted sum of the network's inputs plus a bias, as shown in eq. (42). The output of the network is given by a weighted sum of all neurons and an output bias (eq. (40)). The parameters of the network are the input and output weights (w_{ij} and w_j) and input and output biases (b_j and b). Note that b is not the same as defined in section 3.1 for the OLS algorithm.

$$y = \sum_{j=1}^{n} w_j \phi_j + b = \hat{C}_m \tag{40}$$

$$\phi_j = \frac{2}{1 + \exp\{-2v_j\}} - 1 \tag{41}$$

$$v_j = \sum_{i=1}^{2} w_{ij} u_i + b_j = w_{1j} u_1 + w_{2j} u_2 + b_j \tag{42}$$

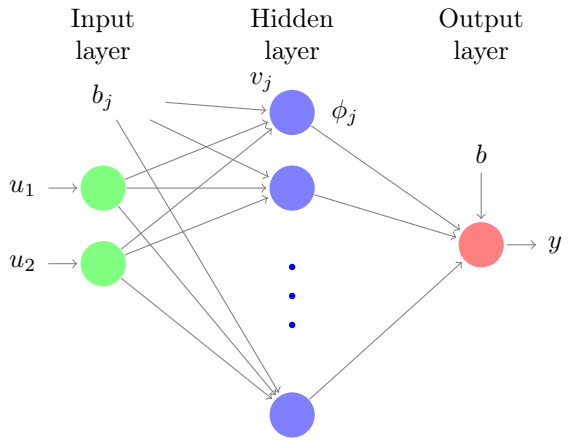


Figure 23: Representation of a FF-NN

4.1 FF-NN learning using the Backpropagation algorithm

In the backpropagation algorithm, the network learns through the help of a first order optimiser such as the Steepest/Gradient Descent. After each epoch, the weight updates for all the data points are obtained using eq. (43), where η is the learning rate. These are averaged to obtain one final update (eq. (44)). The method is called batch gradient descent ([2]), where the batch is the entire training set.

In contrary to the LM algorithm seen in section 3.2, the adaptive learning has a small change in the backpropagation algorithm. If the cost at the current epoch decreases, the changes of the network are accepted and this factor is multiplied by γ; however, if the cost increases, the changes are discarded, keeping the previous values of the weights and the learning rate is multiplied by γ^{-1} (where, again, $\gamma > 1$).

$$\Delta\omega_k^{(q)} = -\eta \frac{\partial E^{(q)}}{\partial \omega_k} \tag{43}$$

$$\Delta\omega_k = \frac{1}{N}\sum_{q=1}^{N} \Delta\omega_k \tag{44}$$

The derivatives of the weights for each data point are presented in eqs. (45) to (48). These minimize the same cost function as mentioned previously in section 3.2 for the LM algorithm. The weight update was already shown by eq. (31).

$$\frac{\partial E}{\partial b} = \frac{\partial E}{\partial e}\frac{\partial e}{\partial y}\frac{\partial y}{\partial b} = e \cdot (-1) \cdot 1 \tag{45}$$

$$\frac{\partial E}{\partial w_j} = \frac{\partial E}{\partial e}\frac{\partial e}{\partial y}\frac{\partial y}{\partial w_j} = e \cdot (-1) \cdot \phi_j \tag{46}$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial e}\frac{\partial e}{\partial y}\frac{\partial y}{\partial \phi_j}\frac{\partial \phi_j}{\partial v_j}\frac{\partial v_j}{\partial w_{ij}} = e \cdot (-1) \cdot w_j \cdot \left(\frac{4\exp\{-2v_j\}}{(1+\exp\{-2v_j\})^2}\right) \cdot (u_i) \tag{47}$$

$$\frac{\partial E}{\partial b_j} = \frac{\partial E}{\partial e}\frac{\partial e}{\partial y}\frac{\partial y}{\partial \phi_j}\frac{\partial \phi_j}{\partial v_j}\frac{\partial v_j}{\partial b_j} = e \cdot (-1) \cdot w_j \cdot \left(\frac{4\exp\{-2v_j\}}{(1+\exp\{-2v_j\})^2}\right) \cdot 1 \tag{48}$$

$$\tag{49}$$

Considering a network with 20 neurons learned with an initial learning rate $\eta = 0.01$ and $\gamma = 10$, the results are presented in figs. 24 and 25, after 100000 epochs. The inner and outer weights were drawn randomly from uniform distribution whose interval is $[-2, 2]$. The inner and outer biases are also initialised randomly from same distribution whose interval goes from 0 to 1. The final value of the training cost is 0.308996, which is bigger than what was obtained through the polynomial or RBF-NN models. It shows that the backpropagation algorithm takes longer to learn the model.

Figure 24: Model performance of the RBF-NN with the number of neurons using the OLS algorithm (IEKF data set)

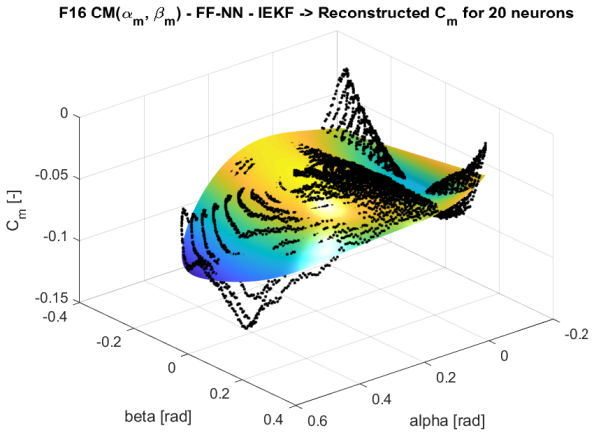


Figure 25: Model performance of the FF-NN with the number of neurons using the LM algorithm (IEKF data set)

To study the sensitivity of this algorithm to initial conditions, the effect of the initial learning rate was evaluated in table 6. This table shows that the initial learning rate does have an effect on the final performance of the network, as it measures the percentage of initial weight updates given to the network in the beginning. The results show that a better performance was achieved when $\eta = 0.1$.

The maximum number of iterations only affects the final performance. For the backpropagation, the model needs more learning iterations to provide a good approximation of the dataset. For instance, it was observed that after 40000 epochs the cost function did present a relevant change to what was obtained at the end of the simulation with 100000 epochs, which leads to the conclusion that the algorithm was stuck in a local minima.

In order to verify the influence of the initial weights of the network in its performance, different intervals were atributed to the uniform distribution used to initialize the weights. Those where inspired from [3] and presented

η [−]	E_{tot} [−]
0.1	0.2734
0.01	0.3901
0.001	0.3923

Table 6: Performance of the FF-NN trained with the backpropagation algorithm for different values of initial damping factor λ (IEKF data set)

in table 7. Taking into account the previous statements, the network was trained up to 40000 iterations and an initial learning rate of 0.1 was chosen. The biases were initialised from the same distribution considering an interval of $[0, 1]$. Note that $fan-in$ is the number of nodes of the previous layer (in this case, input layer) and $fan-out$ is the number of nodes of the following layer (which corresponds to the output layer).

Initialization # (number)	Interval of the uniform distribution [−]	E_{tot} [−]
Baseline	$[-2, 2]$	0.4379
#1	$\left[-\sqrt{\frac{1}{n}}, +\sqrt{\frac{1}{n}}\right]$	1.0963
#2	$\left[-\sqrt{\frac{6}{fan-in+fan-out}}, +\sqrt{\frac{6}{fan-in+fan-out}}\right]$	0.5009
#3	$\left[-4\sqrt{\frac{6}{fan-in+fan-out}}, +4\sqrt{\frac{6}{fan-in+fan-out}}\right]$	2.4417

Table 7: Performance of the FF-NN trained with the backpropgation algorithm for different weight initialisations (IEKF data set)

The results show that none of the given initializations yield better results than the baseline. However, different initial conditions do present different results on the final network. The optimizer searches for different paths towards a minimum value, which in this case seems to be a local minima. Therefore, the algorithm fails approximate the data set.

4.2 FF-NN learning using the Levenberg-Madquardt (LM) algorithm

Similar to section 3.2, a FF-NN can also learn through the LM algorithm using the same training procedure. In this case, the Jacobian matrix is defined in eq. (50). Each FF neuron has 3 weights and the output of the network is influenced by one bias. Hence, there are $3n + 1$ weights to learn, which corresponds to the number of columns of the matrix, or, in other words, the number of derivatives to be computed per data-point. These derivatives can be determined using the expressions deduced in eqs. (51) to (54).

$$J = \begin{bmatrix} \frac{\partial e^{(1)}}{\partial w_{11}} & \cdots & \frac{\partial e^{(1)}}{\partial w_{1n}} & \frac{\partial e^{(1)}}{\partial w_{21}} & \cdots & \frac{\partial e^{(1)}}{\partial w_{2n}} & \frac{\partial e^{(1)}}{\partial b_1} & \cdots & \frac{\partial e^{(1)}}{\partial b_n} & \frac{\partial e^{(1)}}{\partial w_1} & \cdots & \frac{\partial e^{(1)}}{\partial w_n} & \frac{\partial e^{(1)}}{\partial b} \\ \frac{\partial e^{(2)}}{\partial w_{11}} & \cdots & \frac{\partial e^{(2)}}{\partial w_{1n}} & \frac{\partial e^{(2)}}{\partial w_{21}} & \cdots & \frac{\partial e^{(2)}}{\partial w_{2n}} & \frac{\partial e^{(2)}}{\partial b_1} & \cdots & \frac{\partial e^{(2)}}{\partial b_n} & \frac{\partial e^{(2)}}{\partial w_1} & \cdots & \frac{\partial e^{(2)}}{\partial w_n} & \frac{\partial e^{(2)}}{\partial b} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{\partial e^{(N)}}{\partial w_{11}} & \cdots & \frac{\partial e^{(N)}}{\partial w_{1n}} & \frac{\partial e^{(N)}}{\partial w_{21}} & \cdots & \frac{\partial e^{(N)}}{\partial w_{2n}} & \frac{\partial e^{(N)}}{\partial b_1} & \cdots & \frac{\partial e^{(N)}}{\partial b_n} & \frac{\partial e^{(N)}}{\partial w_1} & \cdots & \frac{\partial e^{(N)}}{\partial w_n} & \frac{\partial e^{(N)}}{\partial b} \end{bmatrix} \tag{50}$$

$$\frac{\partial e}{\partial b} = \frac{\partial e}{\partial y}\frac{\partial y}{\partial b} = (-1) \cdot 1 \tag{51}$$

$$\frac{\partial e}{\partial w_j} = \frac{\partial e}{\partial y}\frac{\partial y}{\partial w_j} = (-1) \cdot \phi_j \tag{52}$$

$$\frac{\partial e}{\partial w_{ij}} = \frac{\partial e}{\partial y}\frac{\partial y}{\partial \phi_j}\frac{\partial \phi_j}{\partial v_j}\frac{\partial v_j}{\partial w_{ij}} = (-1) \cdot w_j \cdot \left(\frac{4\exp\{-2v_j\}}{(1+\exp\{-2v_j\})^2}\right) \cdot (u_i) \tag{53}$$

$$\frac{\partial e}{\partial b_j} = \frac{\partial e}{\partial y}\frac{\partial y}{\partial \phi_j}\frac{\partial \phi_j}{\partial v_j}\frac{\partial v_j}{\partial b_j} = (-1) \cdot w_j \cdot \left(\frac{4\exp\{-2v_j\}}{(1+\exp\{-2v_j\})^2}\right) \cdot 1 \tag{54}$$

A FF-NN with 20 neurons is trained with the same initial conditions as mentioned initially in 4.1, where $\lambda = \eta^{-1}$ starts with a value of 100, yielding the results presented in fig. 26. Compared to the backpropagation algorithm, the LM optimizer yields better performance with fewer training iterations, in this case, 3000 epochs.

To analyse the sensitivity of the network to initial conditions, experiments were conducted by changing the number of epochs. λ was also changed while keeping the maximum number of epochs to 3000. Results are presented in tables 8 and 9. Looking at the final costs, they are smaller than obtained for the RBF-NN. This is

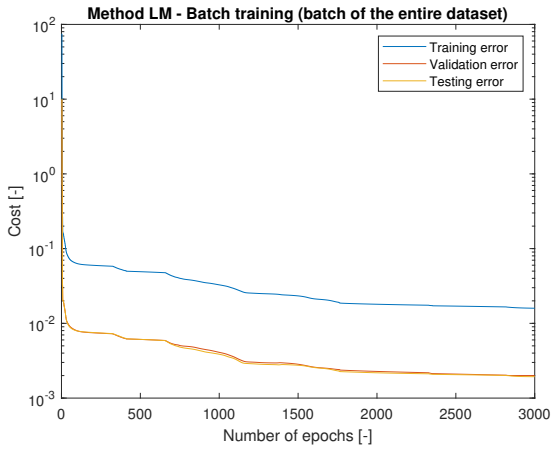


Figure 26: Model performance of the RBF-NN with the number of neurons using the OLS algorithm (IEKF data set)

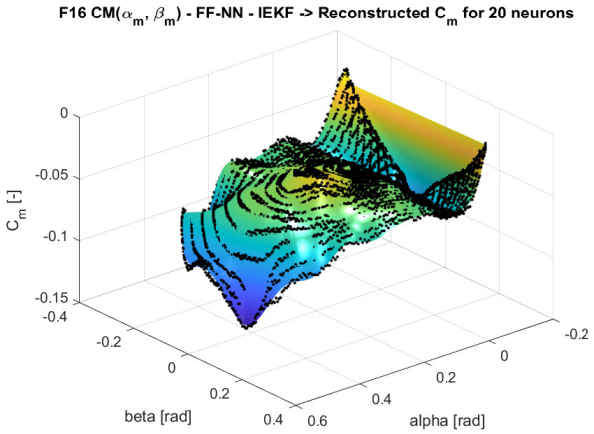


Figure 27: Model performance of the FF-NN with the number of neurons using the LM algorithm (IEKF data set)

due to the global behaviour of sigmoid activation function when compared to the local behaviour of the Radial Basis function. The later yields better results in certain domains of the data set. Increasing the number of epochs yields better results as the training/validation/testing costs decrease. The value of λ that gives lowest cost is again 100 just like for the RBF-NN.

Epoch [−]	E_{train} [−]	E_{val} [−]	E_{test} [−]
3000	0.0167	0.0022	0.0021
6000	0.0154	0.0019	0.0019
9000	0.0154	0.0019	0.0019

Table 8: Performance of the FF-NN with 20 neurons trained with the LM algorithm for different epochs (IEKF data set)

λ [−]	E_{tot} [−]
10	0.0240
100	0.0210
1000	0.0251

Table 9: Performance of the FF-NN trained with the LM algorithm for different values of initial damping factor λ (IEKF data set)

Similar to section 4.1, an experiment to check the influence of the initial weights in the networks performance was also conducted. The network was trained up to 3000 iterations with an initial value for λ of 100 and the biases were drawn from an uniform distribution with the interval $[0, 1]$. The results are presented in table 10.

Initialization # (number)	Interval of the uniform distribution [−]	E_{tot} [−]
Baseline	$[-2, 2]$	0.0210
#1	$\left[-\sqrt{\frac{1}{n}}, +\sqrt{\frac{1}{n}}\right]$	0.0351
#2	$\left[-\sqrt{\frac{6}{fan-in+fan-out}}, +\sqrt{\frac{6}{fan-in+fan-out}}\right]$	0.0201
#3	$\left[-4\sqrt{\frac{6}{fan-in+fan-out}}, +4\sqrt{\frac{6}{fan-in+fan-out}}\right]$	0.0246

Table 10: Performance of the FF-NN trained with the LM algorithm for different weight initialisations (IEKF data set)

Comparing to the baseline, the second initialization provided better results, even though the total cost of both cases are relatively close. Overall, the conclusion of this experiment shows that the network's final performance is sensitive to the initial weights. Moreover, compared to the backpropagation algorithm, the LM optimizer still leads to better results for all 4 cases.

4.3 Number of neurons optimization for the FF-NN

To optimize the number of neurons of the FF-NN the same algorithm described in section 3.3 is employed. Training a network with the LM algorithm network produces better results with less iterations when compared to the backpropagation algorithm. The maximum number of epochs chosen for the task is 400 and the initial value chosen for μ is 100, varying only the number of neurons. The results are depicted in fig. 28.

The optimum number of neurons found under the given conditions is 29 and the total cost associated is 0.068808. The number of neurons is close to what was obtained with the RBF-NN (section 3.3), but the

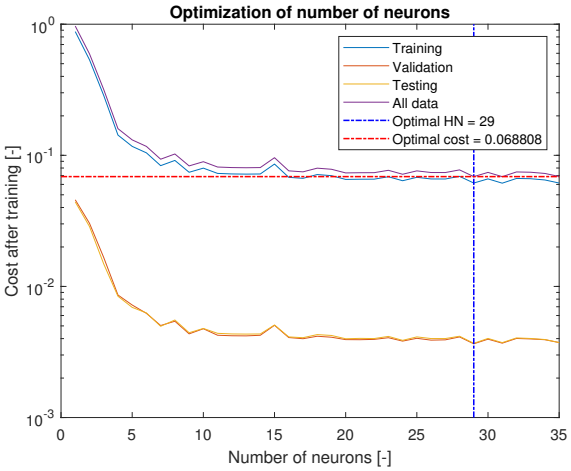


Figure 28: Model performance of the FF-NN with the number of neurons using the OLS algorithm (IEKF data set)

total cost is smaller. This can be explained again through the global approximation nature of the FF-NN. Additionally, the cost function decreases with an increase of the number of neurons. The reason of this trend is the same as discussed before before: the data points are very close to each other, which is difficult for the model to overfit with very few iterations.

4.4 Approximation power comparison of the RBF-NN, FF-NN and polynomial model from LLS algorithm

In this section, a comparison of the approximation power for the model obtained through the OLS algorithm and the trained networks is presented. This is useful to assess which model can be used to obtain a clear mapping of a given data set. Therefore, a polynomial model of $6th$ order is chosen. From section 2.2.1, the total number of parameters to be estimated with the OLS algorithm are given by $\sum_{i=1}^{6+1} i = 1+2+3+4+5+6 = 28$. This also corresponds to the number of neurons chosen for the neural networks.

A RBF-NN and FF-NN are trained using the LM algorithm. The goal is to determine the number of epochs required to achieve following threshold (chosen by the user) for the total cost E_{total}: 0.09. Both networks are initialised with $\lambda = 100$ and the initial conditions are the same used in the beginning of sections 3.2 and 4.2. Both are trained for 400 epochs. The required number of iterations to achieve this value of threshold is 335 for the RBF-NN and 44 for the FF-NN. This is a huge difference and enforces again the global approximation behaviour of the FF-NN compared to the RBF-NN which is mainly local.

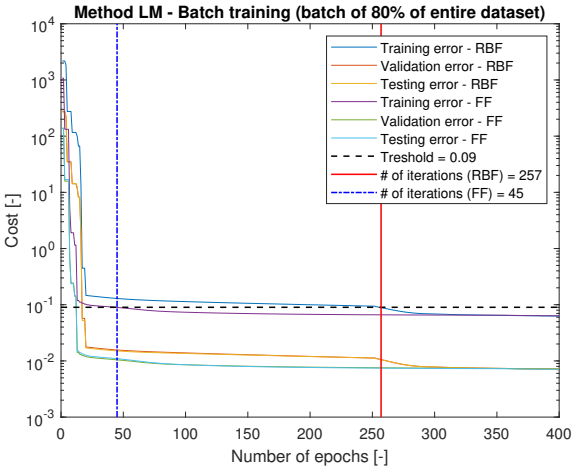


Figure 29: Comparison of the FF-NN and RBF-NN with 28 neurons performance (IEKF data set)

The OLS algorithm can also be used to estimate the parameters of the polynomial model defined in section 2.2.1. The parameters of the polynomial model are determined with data used to train neural networks. The results of all the models are summarised in table 11. In this case, looking at the total cost of the polynomial

model it also ensures the accuracy established by the user. However, as experienced in sections 2.2, 3.1 and 3.3, the accuracy depends on model order and amount of data used in the regression matrix. For instance, if fewer data points are used, then the matrix might not approximate certain regions of the data set.

Model	E_{train}	E_{val}	E_{test}	E_{tot}
RBF-NN	0.0660	0.0082	0.0080	0.0822
FF-NN	0.0474	0.0069	0.0059	0.0600
Polynomial (OLS)	0.0680	0.0082	0.0082	0.0659

Table 11: Training/Validation/Testing/Total cost of different models (IEKF data set)

Overall, comparing all the methods, the FF-NN is the one that yields better results, as its performance is higher even when learned with fewer iterations. However, random division of the dataset or random initialization of the weights can lead to different results. To avoid subjectivity, all the randomness in these choices were removed by adding a seed value to the random number generator ensuring the same results everytime the code is ran.

Conclusions

In this assignment it has been shown the management of the flight data set (or any data set of interest in the aerospace domain) from measurements obtained directly from sensors. State estimation techniques such as Kalman Filters derive other states of the aircraft without the influence of noise. Hence, a mathematical relationship between the derived states can be established by means of polynomial models or non-linear function approximators such as neural networks. To estimate these parameters, least squares algorithm and optimisation methods are used, respectively.

Two neural networks were used: RBF-NN and FF-NN. Their final performance is mainly dependent in the learning algorithm and several conditions chosen initially. RBF-NN are used to approximate data locally whereas FF-NN present a more global behaviour, which is more advantageous. On the other hand, the least squares algorithm can become very complex when lots of data points are used or the model order increases. However, both algorithms can give clear input-output mapping of the data-set and can be used in the system identification of a system, specially in the aerospace domain.

References

[1] E. J. Van Kampen, E. de Weerdt, 2013, "Introduction to Neural Networks", Lecture notes, AE4-312 Modern System Identification of Aerospace Vehicles, Delft University of Technology.

[2] S. Ruder, “An overview of gradient descent optimization algorithms”, 2017. [Online]. Available: `https://arxiv.org/pdf/1609.04747.pdf`

[3] G. Xavier, Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks", 2010.

[4] S. P. Munnoli, A. U. Bapat, "Clustering Algorithms for Radial Basis Function Neural Network", 2013.

[5] C. McCormick, "RBFN Tutorial Part II - Function Approximation", 2015. [Online] Available: `http://mccormickml.com/2015/08/26/rbfn-tutorial-part-ii-function-approximation/`

[6] M. T. Hagan, H. B. Demuth, M. H. Beale, O. De Jesús, "Neural Network Design", 2nd edition.

Appendix

A F-16 aircraft dataset `F16traindata_CMabV_2018.mat`

The F-16 dataset contains 10001 samples of the tuple $(C_m, \alpha, \beta, V, \dot{u}, \dot{v}, \dot{w})$. C_m is the aircraft pitching moment coefficient, α is the angle of attack, β the slideslip angle and $(\dot{u}, \dot{v}, \dot{w})$ are the linear accelerations measured by the aircraft accelerometer.

The aircraft flight envelope is presented in figs. 30 and 31. The first plot involves the relation for raw values of α and β through the 2-D plane. The second one depicts an interpolation plot involving the other 2 variables and the pitching moment, C_m.

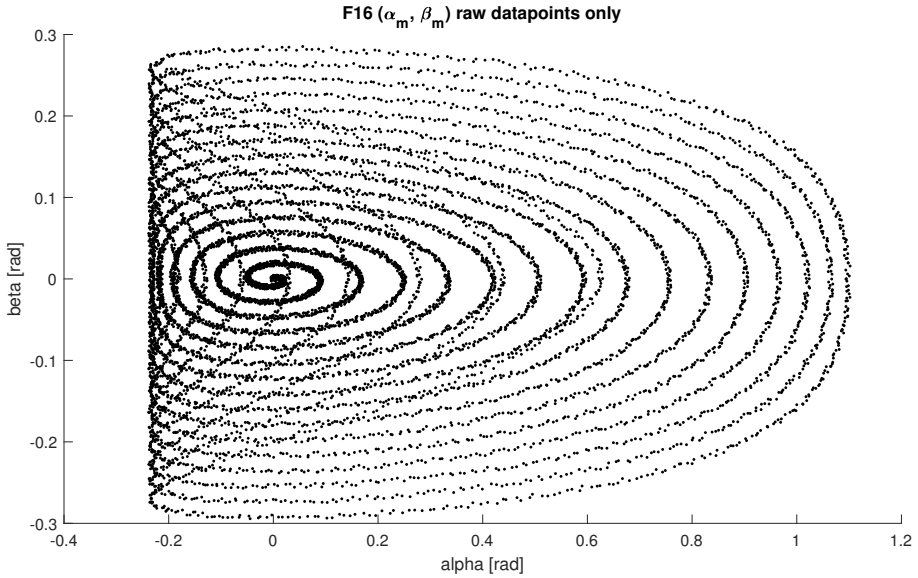


Figure 30: F-16 data points of α and β

This last plot shows the flight envelope of aircraft ensuring a relation between the angle of attack and slideslip angle with the C_m.

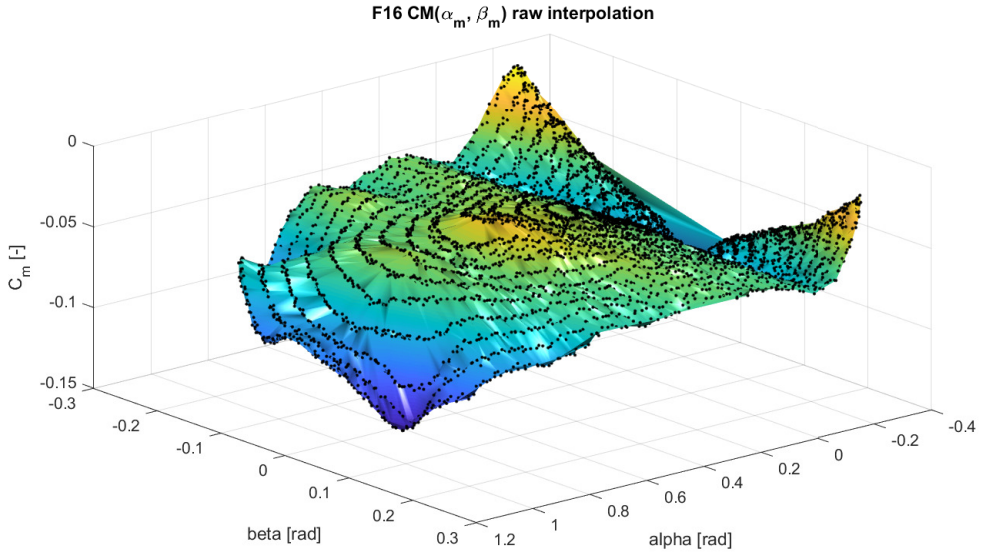
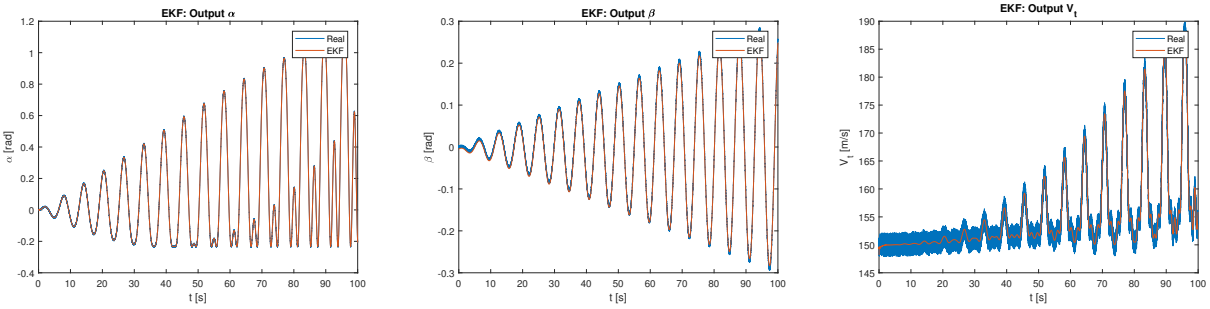


Figure 31: F-16 flight envelope relating C_m with α and β

B Results of EKF (Extended Kalman Filter) and comparison with IEKF

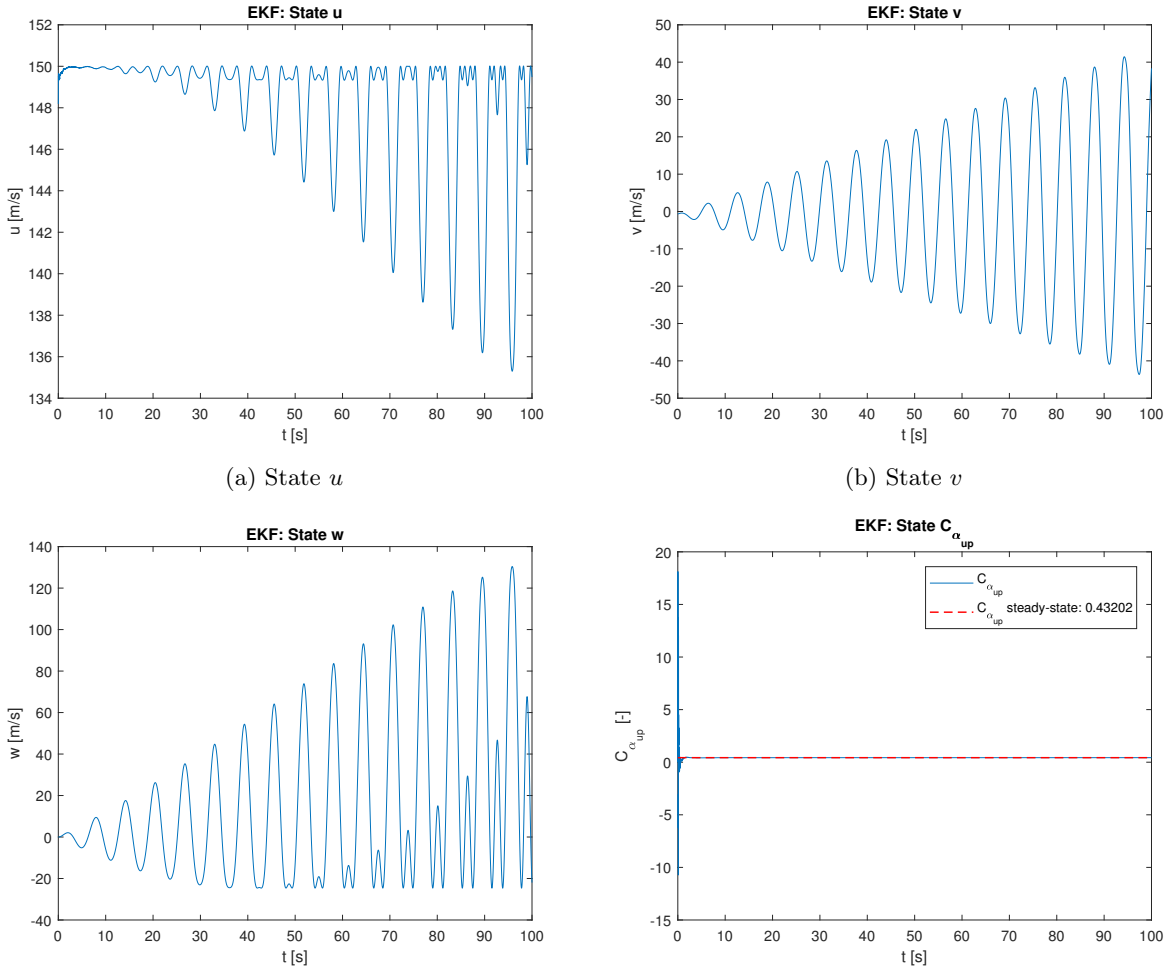
The EKF presents similar results to the IEKF. The noise of the real measurements of α, β and V_t is attenuated in a similar manner in fig. 32.



(a) (b) (c)

Figure 32: Ground-truth and EKF estimation of the measurements: (a) α, (b) β and (c) V_t

The state estimations obtained through this filter are given in fig. 33. A comparison of these states with the ones obtained from the IEKF is shown in fig. 34. Both filters tend to present the same behaviour, however, the EKF is at least 10 times more unstable in the beginning for $C_{\alpha_{up}}$ than for the IEKF. For instance, from fig. 2, this state achieves in the beginning a maximum value of almost 1.6, whereas in fig. 33 it goes to almost 18. However, the steady-state value is the same for both cases.



(a) State u (b) State v

(c) State w (d) State $C_{\alpha_{up}}$

Figure 33: EKF estimation of the states: (a) u, (b) v, (c) w and (d) $C_{\alpha_{up}}$

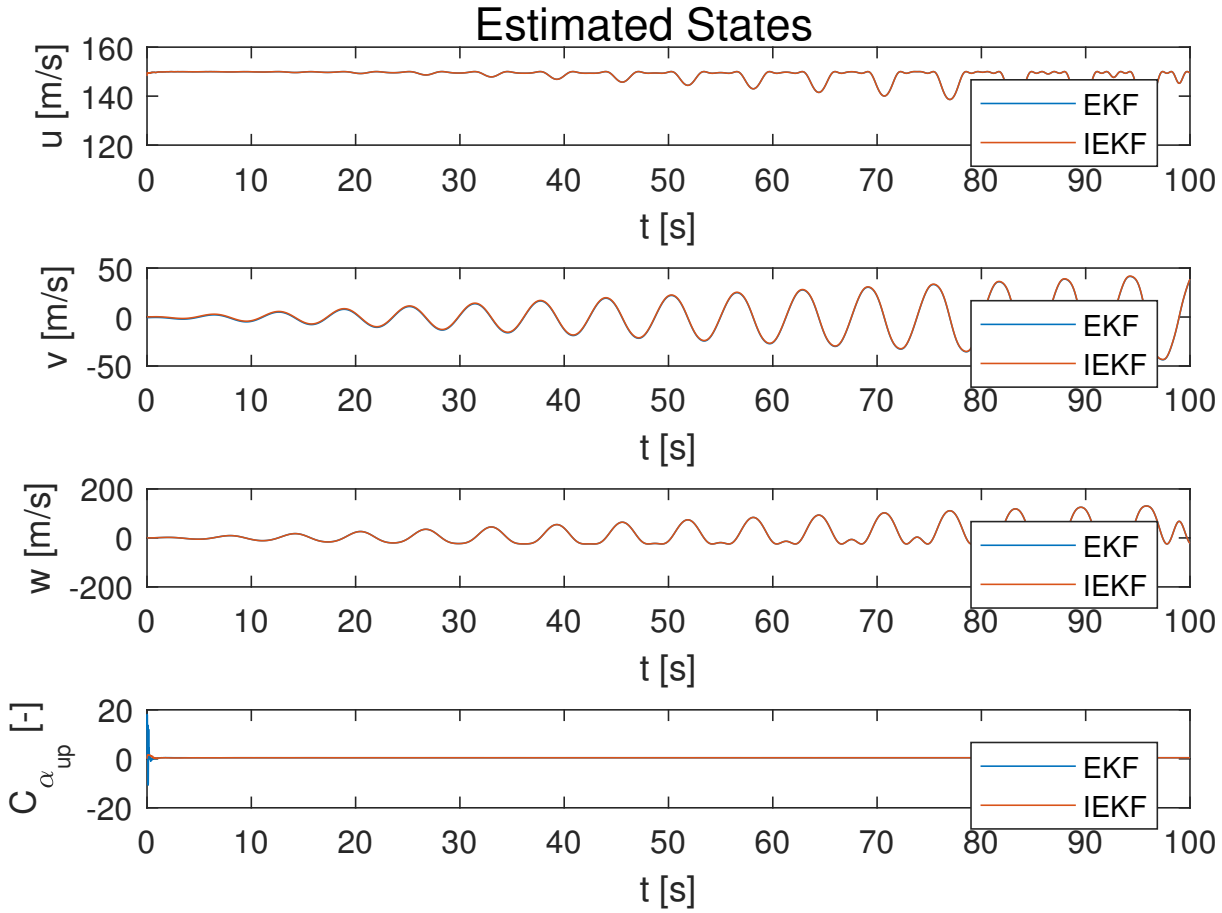
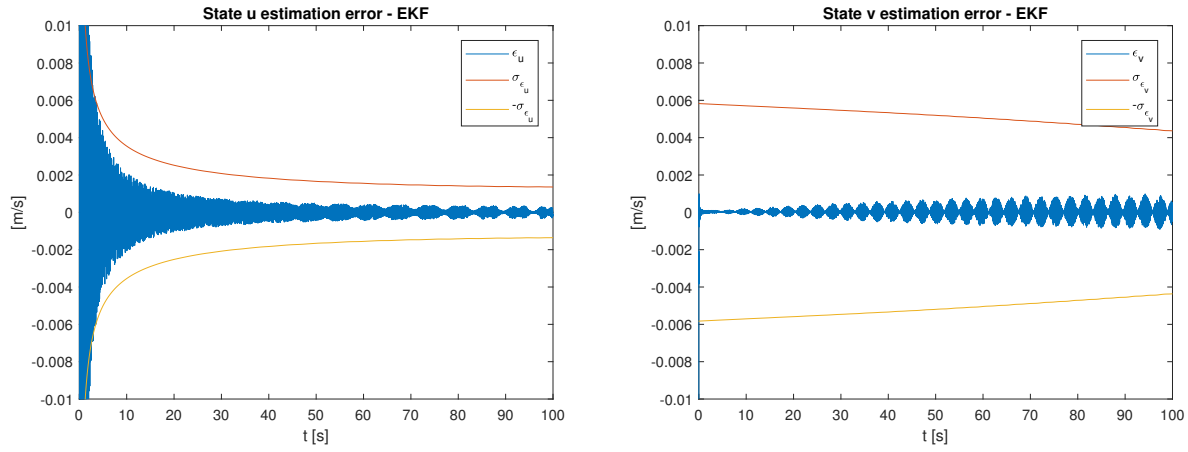


Figure 34: Comparison of state estimations from the EKF and IEKF

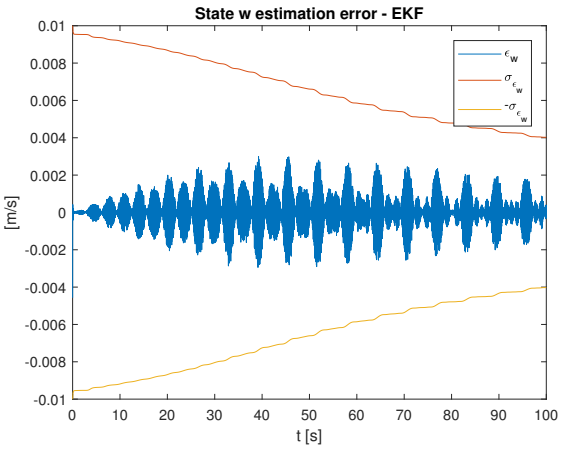
In order to analyse the convergence of the filters, the same state error estimation from eq. (13) is taken into account. The plots are presented in fig. 35 and the errors tend to zero, staying within the bounds of the their standard deviation, which means that filter does converge. This convergence is also confirmed through the observability analysis.



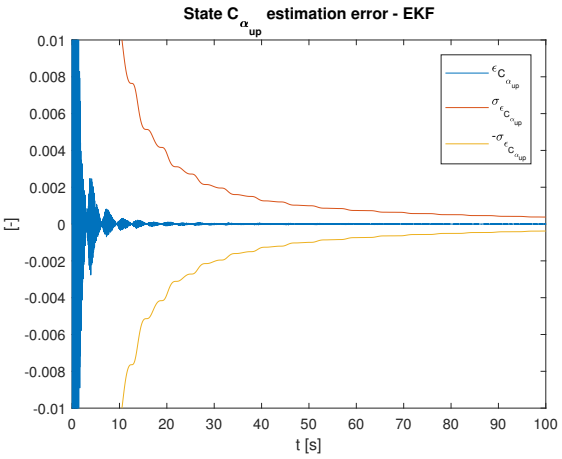
(a) State u

(b) State v

In order to obtain the true value of angle of attack, the same procedure as explained in section 2.1.4 is applied. A plot is obtained in fig. 36. Again, the same conclusions of IEKF can be taken, as here α_{true} is a scaled version and without bias.



(c) State w



(d) State $C_{\alpha_{up}}$

Figure 35: EKF state error estimation: (a) u, (b) v, (c) w and (d) $C_{\alpha_{up}}$

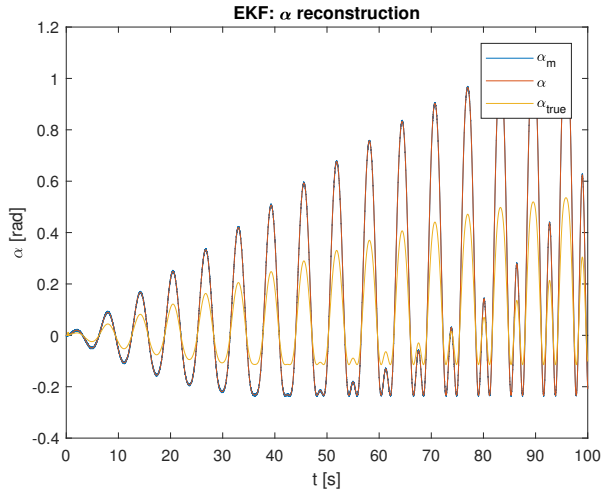


Figure 36: Reconstruction of the angle of attack for the EKF

C Sensitivity analysis of inner weights of the RBF-NN using the OLS algorithm

In order to train the outer weights of the RBF-NN using the OLS algorithm, the inner weights and centers of the RBF neurons should be defined. The location of the centers is defined using the k-means algorithm. An increase of the number of neurons leads smaller clusters in the dataset.

The inner weights are related with the radius of the distribution. Thus, if the number of neurons increases, the radius of the network should decrease, because the neurons start to approximate better the smaller clusters, so that it does not influence the approximation of the data in the adjacent clusters. Thus, taking into account the relation found in eq. (30), the absolute value of the inner weights should increase. To check if this relationship is valid, RBF-NNs from 1 to 100 neurons were trained. For each network, the inner weights are initialized using a uniform distribution that goes $[0, x]$, where x changes for 1 to 30 (in steps of 1). For each model and inner weight initialization, the one that yields the smallest cost is chosen and saved to plot the results in fig. 37.

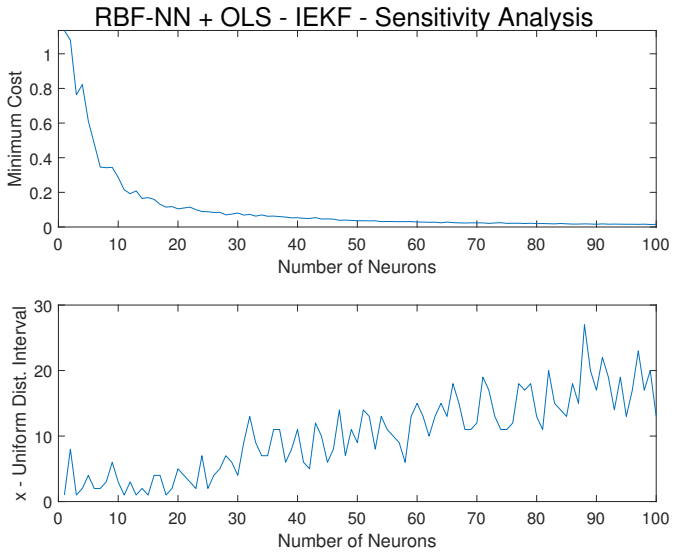


Figure 37: Sensitivity analysis of the inner weights in the RBF-NN trained using the OLS algorithm (IEKF data set)

Looking at the results, the second plot shows the value of x that, for a chosen number of neurons, yield the minimum cost presented in the first plot. For instance, for the case of a model with 100 RBF neurons, the smallest cost using the OLS is obtained for $x = 13$, i.e., for a uniform distribution in the interval $[0, 13]$. The results for this case are given in figs. 38 and 39.

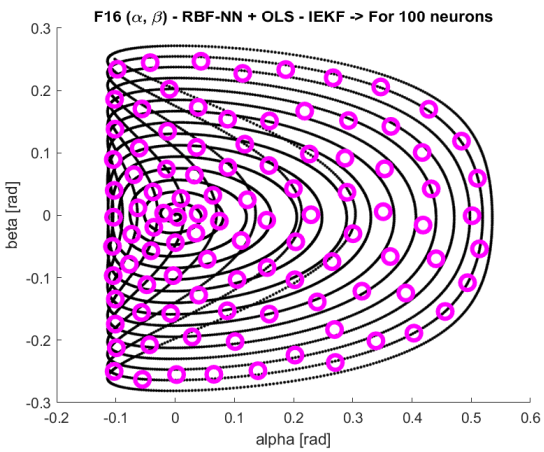


Figure 38: Position of the centers using the k-means clustering algorithm for a RBF-NN with 100 neurons (IEKF data set)

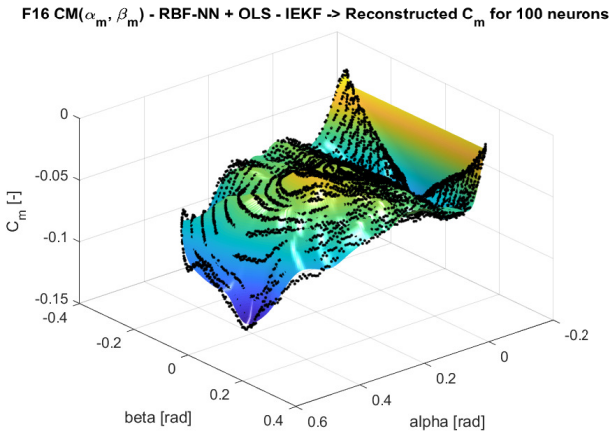


Figure 39: C_m reconstruction (with black dots as the ground-truth) obtained for a RBF-NN with 100 neurons using the OLS algorithm (IEKF data set)

D Learning algorithm

In order to train the neural network, an adaptive learning algorithm (algorithm 1) is chosen, as it optimizes the value of the learning rate at each training iteration. The cost E presented in the algorithm is the training cost. Therefore, the other two costs (validation and testing) are only used to validate the model during and after training.

Algorithm 1 Adaptive Learning algorithm

Initialization: n, μ (or λ), γ, Number of Epochs;
Define percentage of training/validation/testing data;
Create and initialize network with random weights ω_0;
Forward pass the data to determine the outputs of the network;
Compute current cost E_{k-1};
for k < Number of Epochs **do**
  Forward pass the data to determine the outputs of the network;
  Backward pass: determine the update rule $\Delta\omega_k$;
  Update the network: $\omega_k = \omega_{k-1} + \Delta\omega_k$;
  Forward pass the data to determine the outputs of the updated network;
  Determine the cost E_k;
  if $E_k < E_{k-1}$ **then**
    $\mu = \mu \cdot \gamma$ (or $\lambda = \lambda \cdot \gamma^{-1}$);
    Accept the changes by keeping the updated network;
  else if $E_k \leq E_{k-1}$ **then**
    $\mu = \mu \cdot \gamma^{-1}$ (or $\lambda = \lambda \cdot \gamma$); $E_k = E_{k-1}$;
    Keep previous network (at time $k-1$, i.e., before the update);
  end if
  if ($E_k \leq$ Learning goal) **or** (Sum(absolute value of gradients)) $\leq$ gradient$_{\text{min}}$) **or** ($\mu \geq \mu_{max}$) **then**
    Break the loop;
  end if
end for

If one decides to keep the learning rate fixed during training, he/she can opt on choosing a fixed learning algorithm (algorithm 2). The difference comes in the assessment of the cost function: if the cost in the next iteration is lower, then the network is updated, otherwise, the cost in the previous iteration is reused. Therefore, the learning the rate is never updated. Despite the fact this algorithm is not used chosen for the present assignment, scripts were made for the user to assess the different between both learning methods.

Algorithm 2 Fixed Learning algorithm

Initialization: n, μ (or λ), γ, Number of Epochs;
Define percentage of training/validation/testing data;
Create and initialize network with random weights ω_0;
Forward pass the data to determine the outputs of the network;
Compute current cost E_{k-1};
for k < Number of Epochs **do**
  Forward pass the data to determine the outputs of the network;
  Backward pass: determine the update rule $\Delta\omega_k$;
  Update the network: $\omega_k = \omega_{k-1} + \Delta\omega_k$;
  Forward pass the data to determine the outputs of the updated network;
  Determine the cost E_k;
  if $E_k < E_{k-1}$ **then**
    Accept the changes by keeping the updated network;
  else if $E_k \leq E_{k-1}$ **then**
    $E_k = E_{k-1}$;
  end if
  if ($E_k \leq$ Learning goal) **or** (Sum(absolute value of gradients)) $\leq$ gradient$_{\text{min}}$) **or** ($\mu \geq \mu_{max}$) **then**
    Break the loop;
  end if
end for