جامعة خليفة
Khalifa University

Senior Research Project

On a Generalization of Kernel RLS to Nonlinear State-space Systems

Author

Ahmed A. Radhi[1]

(100049714)

Supervisor

Prof. Ibrahim M. Elfadel[2]

Co-Supervisor

Prof. Jorge P. Zubelli[3]

May 8, 2023

[1]Department of Mathematics, Khalifa Univesity, Abu Dhabi P.O. Box 127788, United Arab Emirates (Email: 100049714@ku.ac.ae)

[2]Department of Electrical Engineering and Computer Science, Khalifa Univesity, Abu Dhabi P.O. Box 127788, United Arab Emirates (Email: ibrahim.elfadel@ku.ac.ae)

[3]Department of Mathematics, Khalifa Univesity, Abu Dhabi P.O. Box 127788, United Arab Emirates (Email: jorge.zubelli@ku.ac.ae)

Abstract

In this paper, I provide a clear introduction to show how the recursive least squares (RLS) algorithm can be enhanced by utilizing a kernel function to produce a kernel nonlinear version known as the kernel recursive least squares (KRLS) algorithm. In addition to explaining the algorithm mathematically, the paper tests the KRLS algorithm by applying it in the domain of time-series prediction (TSP). Also, the paper compares the results obtained from the TSP task and compares it with other results reported previously. Furthermore, to extend the idea of using the kernel functions to produce kernel methods, the paper introduces a kernel adaptive filtering algorithm, namely the kernel adaptive autoregressive moving average (KAARMA) algorithm, which is an algorithm that can be used to predict the state variables of a system based on previous inputs and outputs. The paper explains analytically the KAARMA algorithm and compares it with the KRLS algorithm.

Keywords— kernel methods, kernel adaptive filtering (KAF), kernel adaptive autoregressive moving average (KAARMA), least squares, recursive least squares (RLS), kernel recursive least squares (KRLS)

Contents

1 Introduction

1.1 Overview

Kernel methods represent a class of learning algorithms that uses the Mercer kernel function to produce nonlinear versions of familiar linear supervised and unsupervised learning algorithms. For instance, support vector machines (SVMs), which are well-known supervised learning algorithms used in classification and regression tasks, use kernel methods. Based on [1], Vladimir Vapnik, the developer of the SVM algorithm, observed that mapping data into a high-dimensional space may allow problems that are difficult to solve in low dimensions to be easily solved. Therefore, Vapnik introduced the "kernel trick", which implies that input vectors that are applied to a Mercer kernel function can be mapped as an inner product into a high-dimensional space, usually called the Hilbert space $\mathcal{H}$ or the feature space.

One of the main algorithms that the paper will be addressing is a kernel adaptive filtering (KAF) algorithm known as the kernel adaptive autoregressive moving average (KAARMA) algorithm. Similar to the kernel methods, the kernel adaptive filtering algorithms are learning algorithms that use the Mercer kernel function to develop a nonlinear version of the adaptive filtering algorithms. However, compared with the kernel methods, kernel adaptive filtering algorithms have different purposes as they are mainly used in signal processing tasks. In this case, kernel functions are used to produce a nonlinear version of the famous time series model called the autoregressive moving average (ARMA) model. Furthermore, the authors of [2] trained the adaptive filter using the stochastic gradient descent in the feature space; as a result, the algorithm was able to approximate any dynamical or nonlinear time-dependent relationships in the input space. Moreover, the authors compared the KAARMA algorithm to the recurrent neural network (RNN) by using them in solving grammatical inference problems. Due to the results they obtained, the authors observed that the KAARMA algorithm outperforms the first-order and second-order RNNs.

1.2 Contributions

This paper aims firstly, to derive the "linear version" of the KAARMA learning algorithm, the ARMA model, analyze the linear learning algorithm, and show that the learning algorithm is solvable. The paper will provide two ways to show

that the learning algorithm is solvable, particularly the least squares algorithm and the recursive least squares (RLS) algorithm. Secondly, before discussing the "nonlinear version" of the ARMA model, the paper aims to discuss and analyze the "kernelized" or "nonlinear version" of the recursive least squares algorithm, namely the kernel recursive least squares (KRLS) algorithm. Moreover, the KRLS algorithm will be experimentally tested on a famous time series data set that will be introduced later in the paper. Finally, the paper will provide a detailed comparison between both algorithms, the KAARMA algorithm and the KRLS algorithm, to show how they are related or unrelated.

2 ARMA Model

Before addressing the KAARMA algorithm, the "non-kernelized" version of the KAARMA algorithm will be addressed first. Therefore, this section of the paper will discuss the ARMA time-series model thoroughly by providing a detailed derivation of the learning algorithm and analyzing the obtained results.

2.1 Learning Algorithm

Define $x_i \in \mathbb{R}$ to be the internal state, $u_i \in \mathbb{R}$ to be the external input, $y_i \in \mathbb{R}$ to be the output, and $s_i \in \mathbb{R}^2$ to be the state vector, where $i = 0, 1, 2, 3, ...$

Let f, g, $h \in \mathbb{R}$ be the parameters we are estimating.

Then, x_i, y_i, and s_i can be defined

$$x_i = fx_{i-1} + gu_i \tag{1}$$

$$y_i = hx_i \tag{2}$$

$$s_i = \begin{bmatrix} x_i \\ y_i \end{bmatrix} = \begin{bmatrix} fx_{i-1} + gu_i \\ hx_i \end{bmatrix} \tag{3}$$

In the second row of equation (3), we replace x_i by equation (1) producing equation (4).

$$s_i = \begin{bmatrix} fx_{i-1} + gu_i \\ hfx_{i-1} + hgu_i \end{bmatrix} \tag{4}$$

Subsequently, we replace hx_{i-1} by y_{i-1} as indicated in equation (2) resulting in equation (5).

$$s_i = \begin{bmatrix} fx_{i-1} + gu_i \\ fy_{i-1} + hgu_i \end{bmatrix} \tag{5}$$

Obviously, we can take the common parameters f and g outside the vector leading to equation (6).

$$s_i = f \begin{bmatrix} x_{i-1} \\ y_{i-1} \end{bmatrix} + g \begin{bmatrix} 1 \\ h \end{bmatrix} u_i \tag{6}$$

Due to equation (3), we can replace the vector $\begin{bmatrix} x_{i-1} \\ y_{i-1} \end{bmatrix}$ by s_{i-1} resulting in equation (7).

$$\boxed{s_i = fs_{i-1} + g \begin{bmatrix} 1 \\ h \end{bmatrix} u_i} \tag{7}$$

Going back to equation (5), we can also express s_i in the following way:

$$s_i = \begin{bmatrix} x_i \\ y_i \end{bmatrix} = \underbrace{\begin{bmatrix} f & 0 & g \\ 0 & f & gh \end{bmatrix}}_{\Omega^T} \underbrace{\begin{bmatrix} x_{i-1} \\ y_{i-1} \\ u_i \end{bmatrix}}_{\psi(s_{i-1},\, u_i)} \tag{8}$$

Note that x_i in equation (1) represents the first-order ARMA algorithm with external input u_i, and s_i in equation (7) represents the ARMA algorithm operating on two variables.

Next, we define the cost function and error vector as follows, respectively,

$$\epsilon_i = \frac{1}{2} e_i^2 \tag{9}$$

$$e_i = d_i - y_i \tag{10}$$

where d_i is the desired output.

Our goal is to minimize the cost function with respect to the parameters f, g, and h; hence, we apply the gradient descent as shown in the following steps.

First, we calculate $\nabla \epsilon_i$

$$\nabla \epsilon_i = \begin{bmatrix} \frac{\partial \epsilon_i}{\partial f} \\ \frac{\partial \epsilon_i}{\partial g} \\ \frac{\partial \epsilon_i}{\partial h} \end{bmatrix} \tag{11}$$

Note that $y_i = \underbrace{\begin{bmatrix} 0 & 1 \end{bmatrix}}_{\mathbb{I}} \underbrace{\begin{bmatrix} x_i \\ y_i \end{bmatrix}}_{s_i} = \mathbb{I} s_i$, then we calculate the partial derivatives indicated in equation (11).

$$\begin{aligned}
\frac{\partial \epsilon_i}{\partial f} &= \frac{\partial}{\partial f}(\frac{1}{2} e_i^2) && (12)\\
&= e_i \frac{\partial e_i}{\partial f} && (13)\\
&= e_i \frac{\partial (d_i - y_i)}{\partial f} && (14)\\
&= -e_i \frac{\partial y_i}{\partial f} && (15)\\
&= -e_i \left(\frac{\partial y_i}{\partial s_i} \cdot \frac{\partial s_i}{\partial f} \right) && (16)\\
&= -e_i \Bigg(\mathbb{I} \underbrace{\begin{pmatrix} x_{i-1} \\ y_{i-1} \end{pmatrix}}_{s_{i-1}} \Bigg) && (17)\\
&= -e_i \left(\mathbb{I} s_{i-1} \right) && (18)\\
&= -e_i \, y_{i-1} && (19)
\end{aligned}$$

$$\frac{\partial \epsilon_i}{\partial g} = \frac{\partial}{\partial g}(\frac{1}{2}e_i^2) \tag{20}$$
$$= e_i \frac{\partial e_i}{\partial g} \tag{21}$$
$$= e_i \left(\frac{\partial e_i}{\partial y_i} \cdot \frac{\partial y_i}{\partial g} \right) \tag{22}$$
$$= e_i \left(\frac{\partial (d_i - y_i)}{\partial y_i} \cdot \frac{\partial (f y_{i-1} + h g u_i)}{\partial g} \right) \tag{23}$$
$$= e_i \left(-h u_i \right) \tag{24}$$
$$= -e_i \left(h u_i \right) \tag{25}$$

$$\frac{\partial \epsilon_i}{\partial h} = \frac{\partial}{\partial h}(\frac{1}{2}e_i^2) \tag{26}$$
$$= e_i \frac{\partial e_i}{\partial h} \tag{27}$$
$$= e_i \left(\frac{\partial e_i}{\partial y_i} \cdot \frac{\partial y_i}{\partial h} \right) \tag{28}$$
$$= e_i \left(\frac{\partial (d_i - y_i)}{\partial y_i} \cdot \frac{\partial (h f x_{i-1} + h g u_i)}{\partial h} \right) \tag{29}$$
$$= e_i \left(-(f x_{i-1} + g u_i) \right) \tag{30}$$
$$= -e_i \left(f x_{i-1} + g u_i \right) \tag{31}$$
$$= -e_i \left(x_i \right) \tag{32}$$
$$= -e_i \left(\frac{y_i}{h} \right) \tag{33}$$

Therefore, $\nabla \epsilon_i$ can now be written as shown in equation (34):

$$\nabla \epsilon_i = \begin{bmatrix} \frac{\partial \epsilon_i}{\partial f} \\ \frac{\partial \epsilon_i}{\partial g} \\ \frac{\partial \epsilon_i}{\partial h} \end{bmatrix} = -e_i \begin{bmatrix} y_{i-1} \\ h u_i \\ \frac{y_i}{h} \end{bmatrix} \tag{34}$$

Now, the gradient descent will have the following form:

$$\begin{bmatrix} \Delta f_i \\ \Delta g_i \\ \Delta h_i \end{bmatrix} = -\eta \begin{bmatrix} \frac{\partial \epsilon_i}{\partial f} \\ \frac{\partial \epsilon_i}{\partial g} \\ \frac{\partial \epsilon_i}{\partial h} \end{bmatrix} \tag{35}$$

Finally, the learning algorithm can be formulated as the following explicit numerical scheme:

$$\boxed{\begin{aligned} f_i &= f_{i-1} + \eta\, e_{i-1}\, y_{i-1} \\ g_i &= g_{i-1} + \eta\, e_{i-1}\, h_{i-1}\, u_i \\ h_i &= h_{i-1} + \eta\, e_{i-1}\, \big(\frac{y_{i-1}}{h_{i-1}}\big) \end{aligned}} \tag{36}$$

As illustrated in the figure below, we can use the learning algorithm in equation (36) to estimate the system parameters (f, g, h) in real-time. In other words, the system parameters will be updated at every time instant based on the previous values, and the current or previous data.

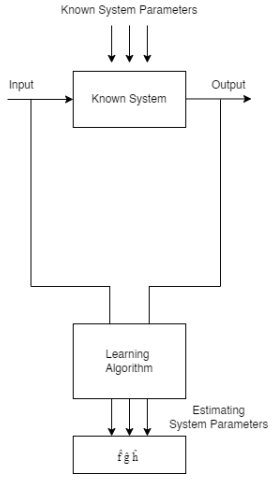


Figure 1: Illustrates How the Learning Algorithm Works.

A MATLAB code was implemented based on the derived learning algorithm in equation (36). As shown in Appendix 6.1, the MATLAB code starts by simulating random data and follows that by estimating the system parameters, in which the parameters were set as follows:

$$f = 0.5$$
$$g = 1.0$$
$$h = 2.0$$

Furthermore, the MATLAB code estimates the system parameter by iterating through 100 iterations. Based on Figure 2, it was concluded that the estimates are not stable, because when comparing the values obtained at different iterations, it was observed that estimates are changing significantly. Moreover, the estimates of f, g, and h obtained at the 100th iteration were the following:

$$f = 0.57268$$
$$g = 1.1701$$
$$h = 1.4245$$

Comparing the estimates of f and g with the exact values, error percentages of 14.5% and 17.0% were obtained for f and g, respectively, which are slightly large. In addition, the estimated h is far from the exact value with an error percentage of 28.8%.

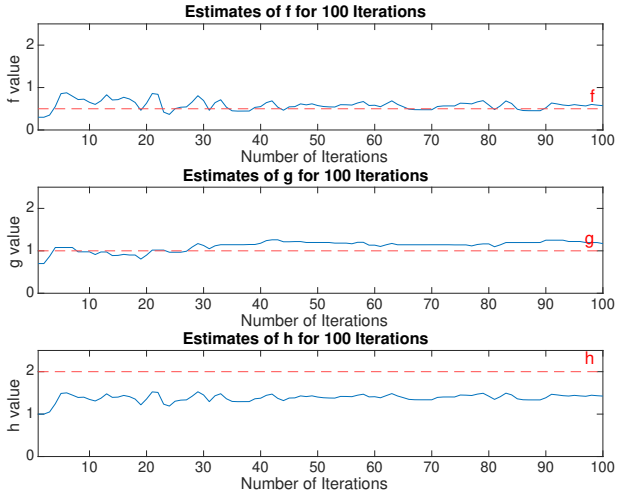


Figure 2: Comparing the Estimates (blue line) of f, g, and h with the Exact Value (red dashed-line) Using 100 Iterations.

Due to the instability observed in the expected values, other algorithms or optimization methods should be used to solve the learning algorithm and estimate the parameters.

2.2 Parameters Estimation

As it was used in the previous part, the error percentage will be applied as a metric to evaluate the performance of each method by comparing the exact parameters to their estimated values.

Furthermore, the error percentage can be calculated by using the following formula:

$$\boxed{\%\,\text{error} = \frac{|\,\text{estimated value} - \text{exact value}\,|}{\text{exact value}} \times 100\%} \tag{37}$$

In addition, to estimate the unknown parameters using the other methods, the least squares and the recursive least squares (RLS), equation (2) will be reformulated into an equation with two unknowns (f and β) instead of three unknowns (f, g, and h).

To reformulate equation (2), equation (1) will be replaced in equation (2), and the following equations will demonstrate the remaining process thoroughly.

$$y_i = h\Big(fx_{i-1} + gu_i\Big) \tag{38}$$

$$= f\underbrace{hx_{i-1}}_{y_{i-1}} + hgu_i \tag{39}$$

$$= fy_{i-1} + \underbrace{hg}_{\beta=hg} u_i \tag{40}$$

$$y_i = fy_{i-1} + \beta u_i \tag{41}$$

Now, equation (41) has two unknowns, namely f and β, where $\beta = hg$. Therefore, the estimation methods should be applied to estimate f and β that solve equation (41).

2.2.1 Least Squares Method

Equation (41) will result in n-1 equations for n $\in \mathbb{N}$ representing the number of iterations.

$$\begin{aligned} y_2 &= fy_1 + \beta u_2 \\ &\vdots \\ y_n &= fy_{n-1} + \beta u_n \end{aligned} \tag{42}$$

Hence, the system of equations in equation (42) can be written as the matrix equation $Ax = b$.

$$\overbrace{\underbrace{\begin{bmatrix} y_1 & u_2 \\ \vdots & \vdots \\ y_{n-1} & u_n \end{bmatrix}}_{(n-1)\times 2}}^{A} \overbrace{\underbrace{\begin{bmatrix} f \\ \beta \end{bmatrix}}_{2\times 1}}^{x} = \overbrace{\underbrace{\begin{bmatrix} y_2 \\ \vdots \\ y_n \end{bmatrix}}_{(n-1)\times 1}}^{b} \tag{43}$$

The system of equations in equation (43) is an over-determined system. Based on [3], a system of equations is considered over-determined when the number of equations is greater than the number of unknowns.

For instance, for $n = 4$, the following system of equations will be obtained. It is obvious that the system of equations has 3 equations, but 2 unknowns only.

$$\overbrace{\underbrace{\begin{bmatrix} y_1 & u_2 \\ y_2 & u_3 \\ y_3 & u_4 \end{bmatrix}}_{3\times 2}}^{A} \overbrace{\underbrace{\begin{bmatrix} f \\ \beta \end{bmatrix}}_{2\times 1}}^{x} = \overbrace{\underbrace{\begin{bmatrix} y_2 \\ y_4 \\ y_4 \end{bmatrix}}_{3\times 1}}^{b} \tag{44}$$

Therefore, for the error vector $\boldsymbol{e} = A\boldsymbol{x} - \boldsymbol{b}$, the least squares method can be used to find the vector $\boldsymbol{x}$ that minimizes the norm of the error vector defined in the following equation

$$||A\boldsymbol{x} - \boldsymbol{b}||^2 = (A\boldsymbol{x} - \boldsymbol{b})^{\mathrm{T}}(A\boldsymbol{x} - \boldsymbol{b}) \tag{45}$$

such that

$$\boldsymbol{x} = (A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}}\boldsymbol{b} \tag{46}$$

As shown in Appendix 6.2.1, a MATLAB function that takes two inputs was written. The function generates the data by iterating through a given number of iterations and estimates the unknown parameters based on a given number of data points.

Furthermore, a MATLAB program code is provided in Appendix 6.2.2. The code is written to go through 101 iterations four times and compare how estimating the unknown parameters using a higher or lower number of data points differs.

Data Points	f **value**	f **Error %**	β **value**	β **Error %**
25	0.48949	2.103	2.1523	7.6165
50	0.49054	1.892	2.0758	3.7911
75	0.49331	1.3388	2.0507	2.5351
100	0.50439	0.87887	1.9905	0.47517

Table 1: Estimates of f and β Using Different Number of Data Points

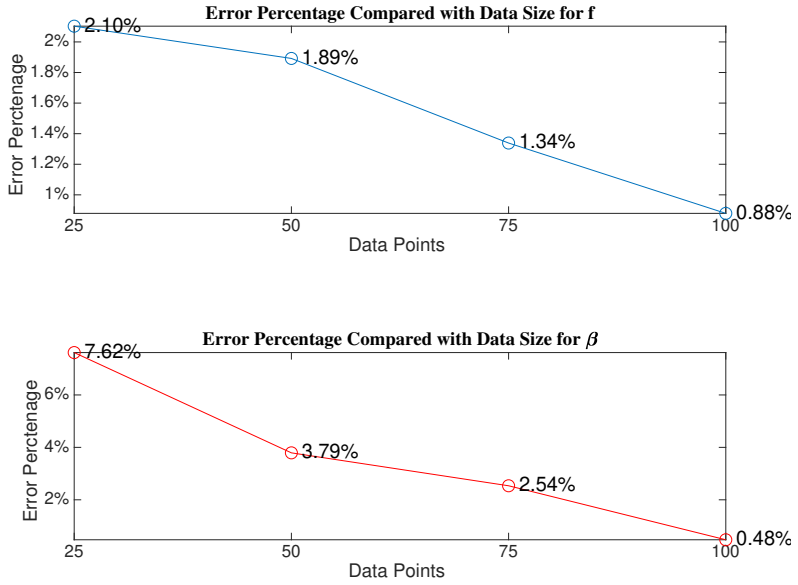


Figure 3: Error Percentages of f and β Estimates for Different Number of Data Points

Based on Table 1 and Figure 3, several findings can be realized. First, it was observed that usually higher number of data points result in a lower error percentage. Second, compared with Figure 2, lower error percentages were achieved by the least squares method. Third, more accurate estimations were obtained using the least squares method than the estimations reported in Figure 2.

Furthermore, the MATLAB code generates four different simulations of data and estimates the unknown paramters for each simulation of data using a given number of data points. Due to that, another run of the code may result in an error percentage achieved by a lower number of data points that is lower than an error percentage achieved by a higher number of data points. However, in all cases, it was observed that the estimations obtained are more accurate than those in Figure 2.

2.2.2 Recursive Least Squares (RLS) Method

As the name of the method may suggest, the recursive least squares (RLS) method is a recursive method that updates the $\boldsymbol{x}$ vector that minimizes the norm of the error vector shown in equation (45) whenever a new data point is added.

As [3] explained, the idea of the RLS method is that it updates the previous estimate by adding a correction term based on the newly added data point resulting in a new estimate.

To explain the RLS method mathematically, for k $\in \mathbb{N}$, denote A_k as the A matrix with k elements and $\boldsymbol{b}_k$ as the $\boldsymbol{b}$ vector with k elements. Then A_k, $\boldsymbol{b}_k$, and $\boldsymbol{x}_k$ can be defined as follows:

$$A_k = \begin{bmatrix} y_1 & u_2 \\ \vdots & \vdots \\ y_k & u_{k+1} \end{bmatrix}, \quad \boldsymbol{x}_k = \begin{bmatrix} f \\ \beta \end{bmatrix}, \quad \boldsymbol{b}_k = \begin{bmatrix} y_2 \\ \vdots \\ y_{k+1} \end{bmatrix} \tag{47}$$

If a new data point (k+1) is added, then A_{k+1} and $\boldsymbol{b}_{k+1}$ can be defined as follows:

$$A_{k+1} = \begin{bmatrix} A_k \\ a_{k+1}^{\mathrm{T}} \end{bmatrix}, \quad \boldsymbol{b}_{k+1} = \begin{bmatrix} b_k \\ y_{k+2} \end{bmatrix} \tag{48}$$

where a_{k+1} and y_{k+2} represent the new data point, and a_{k+1} can be expressed as the following equation:

$$a_{k+1} = \begin{bmatrix} y_{k+1} \\ u_{k+2} \end{bmatrix} \tag{49}$$

Furthermore, the inverse matrix P_k can be defined as the following equation:

$$P_k = \Big[A_k^{\mathrm{T}} \, A_k \Big]^{-1} \tag{50}$$

And once a new data point (k+1) is added, the inverse matrix updates as expressed in the equation below:

$$P_{k+1} = P_k - P_k a_{k+1} \Big[a_{k+1}^{\mathrm{T}} P_k a_{k+1} + 1 \Big]^{-1} a_{k+1}^{\mathrm{T}} P_k \tag{51}$$

Obviously, the $\boldsymbol{x}$ vector also gets updated, and $\boldsymbol{x}_{k+1}$ can be written as the following equation:

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \underbrace{P_{k+1} a_{k+1}}_{\text{gain matrix}} (y_{k+2} - a_{k+1}^{\mathrm{T}} \boldsymbol{x}_k) \tag{52}$$

The gain matrix $P_{k+1}a_{k+1}$ can be written as expressed in the following equation:

$$\begin{aligned} K_{k+1} &= P_{k+1} a_{k+1} \\ &= P_k a_{k+1} \Big[a_{k+1}^{\mathrm{T}} P_k a_{k+1} + 1 \Big]^{-1} \end{aligned} \tag{53}$$

Finally, equation (51) can be reformulated resulting in the following equation:

$$P_{k+1} = P_k - K_{k+1} \, a_{k+1}^{\mathrm{T}} P_k \tag{54}$$

A MATLAB code that can be found in Appendix 6.3.2 is written to iterate through 101 iterations, assumes that $\boldsymbol{x}_1 = 0$, and estimates $\boldsymbol{x}_k$ for k $\in [2, 101]$. The code is based on the "RLSE_Online" [3] function attached in Appendix 6.3.1. Furthermore, Table 2 shows below the estimated values f and β for $\boldsymbol{x}_{26}$, $\boldsymbol{x}_{51}$, $\boldsymbol{x}_{76}$, and $\boldsymbol{x}_{101}$.

k	f value	f Error %	β value	β Error %
26	0.48986	2.028	2.1498	7.4903
51	0.49348	1.3035	2.0786	3.931
76	0.49526	0.94742	2.0521	2.6032
101	0.49624	0.75272	2.0373	1.8651

Table 2: Estimates of f and β for Different Values of k

Below, Figure 4 compares the error percentages for the estimated parameters for $\boldsymbol{x}_{26}$, $\boldsymbol{x}_{51}$, $\boldsymbol{x}_{76}$, and $\boldsymbol{x}_{101}$, and shows how the error percentages decreased as the value of k increased. Moreover, comparing the error percentages in Table 2 with the values reported in Table 1, it can be observed that the error percentages are almost identical in most cases. This observation is due to the fact that both methods are based on the least squares method; however, the RLS method updates the estimates in real-time whenever a new data point arrives.

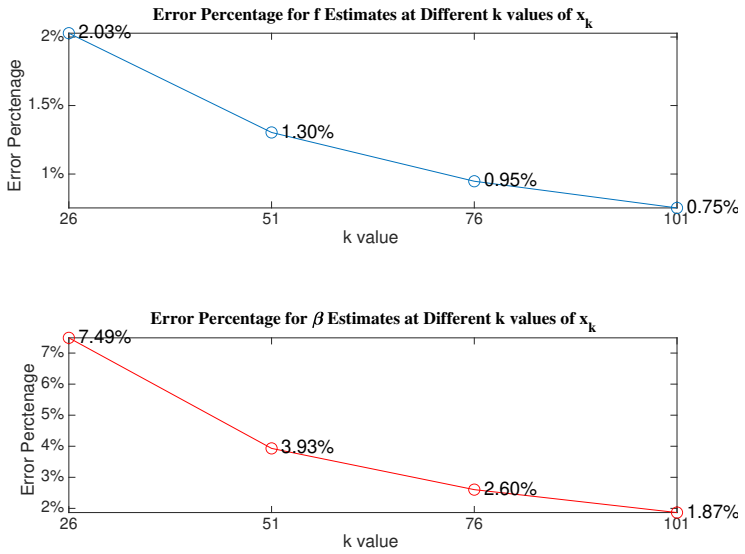


Figure 4: Error Percentages of f and β Estimates for Different Number of k Values

Furthermore, Figure 5 clearly displays the process of how the RLS method estimates the f and β as the number of iterations increases and compares the estimates to the exact values of the parameters.

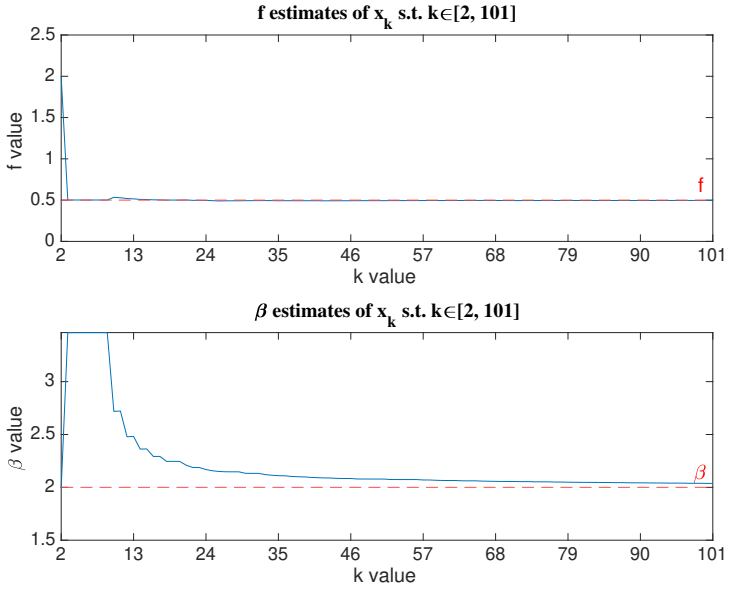


Figure 5: Comparing the Estimates (blue line) of f and β with the Exact Value (red dashed-line) Using 101 Iterations.

3 KRLS Algorithm

3.1 Learning Algorithm

The previous section introduced the recursive least squares (RLS) method and explained how the RLS method can be used to estimate unknown parameters. This section of the paper will introduce the "kernelized version" of the RLS and will show how "non-linearizing" the RLS can be helpful in addressing nonlinear problems.

In this section of the paper, a kernel function will be used to produce a nonlinear version of the RLS introduced in Section 2.2.2, namely the kernel recursive least squares (KRLS) algorithm. Usually, solutions obtained by the kernel methods are of the following form

$$\hat{f}(\mathbf{x}) = \sum_{i=1}^{\ell} \alpha_i k(\mathbf{x}_i, \mathbf{x}) \tag{55}$$

where $\{\mathbf{x}_i\}_{i=1}^{\ell}$ are the training data points, $k(\cdot, \cdot)$ is the kernel function, and

α_i are the coefficients. There are different examples of kernel functions, and two well-known and frequently used kernel functions are the polynomial kernel and the Gaussian kernel. The kernel function that will be used throughout this paper is the Gaussian kernel, and it can be expressed as shown in equation (56).

$$k(\mathbf{x}, \mathbf{x}') = \exp \frac{-(\mathbf{x}-\mathbf{x}')^{\mathrm{T}}(\mathbf{x}-\mathbf{x}')}{2\sigma^2} \tag{56}$$

As [4] explained, some kernel methods obtain regularization by using sparsification, which is the process that results in the disappearance of many of the coefficients (α_i) in the solution defined in equation (55). There are several approaches for sparsification, but not all approaches are suitable for real-time online algorithms. The sparsification approach that the KRLS uses is an online constructive sparsification. The online sparsification approach compares the samples, and the samples that cannot be represented by a linear combination of the previously added samples are added into the kernel representation. In more details, after mapping the inputs into the feature space or the Hilbert space $\mathcal{H}$, that sparsification approach determines if the feature vector of a specific sample is approximately dependent on the other samples or not. If the feature vector is not approximately dependent on the previous samples, then it will be added to a dictionary $\mathcal{D}$, and its coefficient will be added to $\hat{f}$.

The solution of the kernel methods is previously expressed in equation (55); however, mapping the input vectors into the Hilbert space produces the following expression

$$\hat{f}(\mathbf{x}) = \sum_{i=1}^{\ell} \alpha_i k(\phi(\mathbf{x}_i), \phi(\mathbf{x})) \tag{57}$$

In addition, the authors of [4] demonstrated more clearly how the sparsification process can be implemented mathematically. For input-output pairs $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots\}$, such that $x_i \in \mathcal{X}$ and $y_i \in \mathbb{R}$. At time step t, previous $t-1$ training samples $\{\mathbf{x}_i\}_{i=1}^{t-1}$ have been observed, and a subset of the training samples is admitted into the dictionary $\mathcal{D}_{t-1} = \{\mathbf{x}_j\}_{j=1}^{m_{t-1}}$, where the feature vectors of the admitted training samples $\{\phi(\tilde{\mathbf{x}}_j)\}_{j=1}^{m_{t-1}}$ are linearly independent. If we considered an additional training sample $\mathbf{x}_t$, then it can be decided whether the feature vector $\phi(\mathbf{x}_t)$ is linearly independent or not by testing if it is approximately linearly dependent on the dictionary vectors or not. If the feature vector

is not *approximately linearly dependent*, then it will be added to the dictionary. As a result, the training samples $\mathbf{x}_1, \ldots, \mathbf{x}_t$ can be approximated as a linear combination of the vectors in the dictionary $\mathcal{D}_t$.

Furthermore, the authors of [4] introduced the approximate linear dependence (ALD) condition and they stated that if the following condition

$$\delta_t \stackrel{\text{def}}{=} \min_{\boldsymbol{a}} \left\| \sum_{j=1}^{m_{t-1}} a_j \phi(\tilde{\mathbf{x}}_j) - \phi(\mathbf{x}_t) \right\| \leq \nu \,, \tag{58}$$

where ν is the accuracy parameter that determines the level of sparsity and $a_j = \tilde{\mathbf{K}}_{j-1}\tilde{\mathbf{k}}_{j-1}(\mathbf{x}_j)$ such that $\tilde{\mathbf{K}}$ denotes the dictionary kernel matrix and $\tilde{\mathbf{k}}_{j-1}(\mathbf{x}_t) = \tilde{\mathbf{K}}_{j-1}^{\mathrm{T}}\boldsymbol{a}_j$, was satisfied by finding coefficients $\boldsymbol{a} = (a_1, \ldots, a_{m_{t-1}})^{\mathrm{T}}$ that satisfies the condition, then adding the training sample $\mathbf{x}_t$ to the dictionary will be avoided.

Generally, at each time step t, either one of two scenarios is faced:

1. If the feature vector $\phi(\mathbf{x}_t)$ is ALD on $\mathcal{D}_{t-1}$ such that $\delta_t \leq \nu$, then $\mathbf{x}$ is not added to the dictionary and $\mathbf{k}_{tt} = k(\mathbf{x}_t, \mathbf{x}_t)$. As a result, $\mathcal{D}_t = \mathcal{D}_{t-1}$, $m_t = m_{t-1}$, and $\tilde{\mathbf{K}}_t = \tilde{\mathbf{K}}_{t-1}$, in which $\mathcal{D}$ denotes the dictionary, m denotes the dictionary size.

2. If the feature vector $\phi(\mathbf{x}_t)$ is not ALD on $\mathcal{D}_{t-1}$ such that $\delta_t > \nu$, then $\mathbf{x}$ is added to the dictionary. Consequently, $\mathcal{D}_t = \mathcal{D}_{t-1} \cup \{\mathbf{x}_t\}$, $m_t = m_{t-1} + 1$, and $\tilde{\mathbf{K}}_t$ grows as expressed in the following formula.

$$\tilde{\mathbf{K}}_t = \begin{bmatrix} \mathbf{K}_{t-1} & \tilde{\mathbf{k}}_{t-1}(\mathbf{x}_t) \\ \tilde{\mathbf{k}}_{t-1}(\mathbf{x}_t)^{\mathrm{T}} & k_{tt} \end{bmatrix} \tag{59}$$

where $\tilde{\mathbf{k}}_{t-1}(\mathbf{x}_t) = \tilde{\mathbf{K}}_{t-1}^{\mathrm{T}}\boldsymbol{a}_t$, and $\mathbf{k}_{tt} = k(\mathbf{x}_t, \mathbf{x}_t)$

Furthermore, when comparing the dictionary kernel matrix $\tilde{\mathbf{K}}$ to the gain matrix K in the RLS, the dictionary kernel matrix is not exactly the same as the gain matrix. However, the dictionary kernel matrix plays a similar role.

3.2 Example

The authors of [4] experimentally tested the KRLS algorithm in time-series prediction (TSP). They applied the algorithm to the laser time-series data set

from the Santa Fe time-series competition[5]*. The authors used a training data set that consisted of 1000 samples as shown in Figure 6, and they predicted the next 100 samples of the time-series, in which those 100 samples represented the test data set. Due to the three intensity collapses that occur in the training data set shown in Figure 6 below, the chaotic laser time-series data set is considered to be difficult to predict.

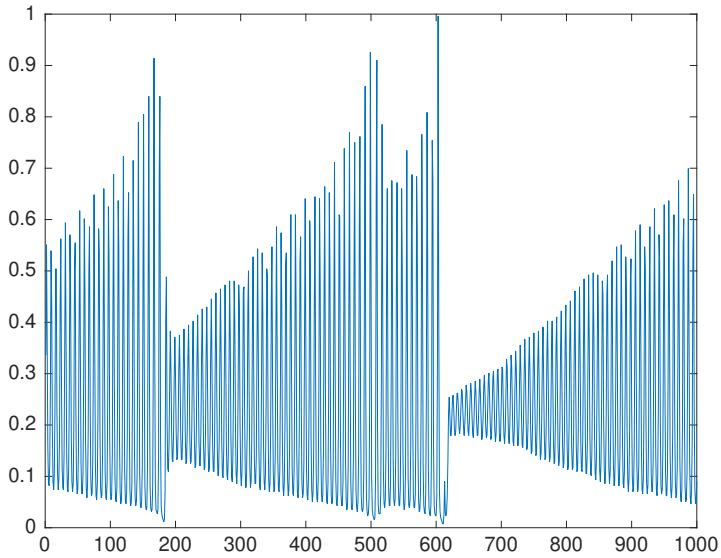


Figure 6: Laser Santa Fe Competition Time-Series Training Data Set

To compare their results to the competition results[6], the authors of [4] calculated the normalized mean squared error (NMSE) they achieved by the KRLS. [6] reported that out of 14 competitors, only two competitors obtained significant prediction accuracy, in which they achieved NMSE of 0.028 and 0.080, respectively. Furthermore, the winning entry obtained an NMSE of 0.028 by using a 'highly specialized neural network architecture adapted by a temporal version of the back propagation algorithm'[4]. On the other hand, the authors of [4] achieved an NMSE of 0.026 using the KRLS algorithm, which is slightly better than that achieved by the winner in the Santa Fe competition.

This paper also implemented the KRLS algorithm on the same data set, and compared the achieved results to those reported in [4] and [6]. The MATLAB

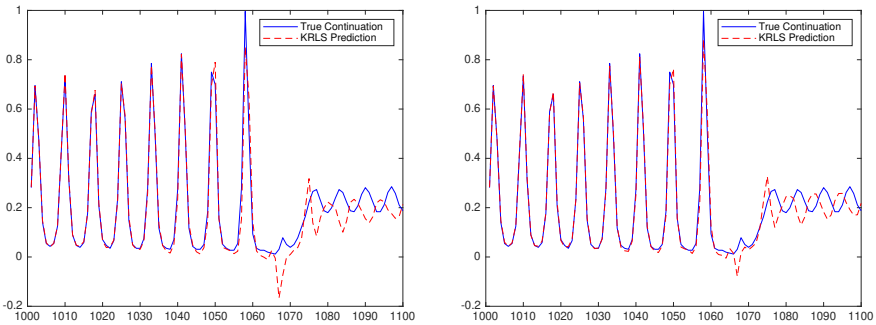
*http://www-psych.stanford.edu/ãndreas/Time-Series/SantaFe.html

codes[‡]in Appendix 6.4 used the KRLS algorithm to predict the 100 data points of the time-series. Furthermore, the code utilized the same parameters that the authors of [4] used in their approach. The following parameters were utilized:

$$\sigma = 0.9$$
$$\nu = 0.01$$
$$n = 40$$

in which they represent the kernel width, the ALD threshold parameter, and the auto-regressive window size, respectively.

Moreover, the MATLAB code addressed the time-series prediction task in two ways. First, normally, the MATLAB code used training set composed of 1000 data points shown in Figure 6 to predict the subsequent 100 data points, and achieved a NMSE of 0.069428 as shown in Figure 7. Comparing the achieved NMSE with the previously reported NMSEs, it can be observed that the NMSE achieved by the attached MATLAB code is worse than the NMSE obtained by the authors of [4] and the first competitor. However, the obtained NMSE is slightly better than the NMSE achieved by the second competitor.



(a) NMSE = 0.069428 (b) NMSE = 0.042877

Figure 7: Comparing the 100 Predicted Data Points Using KRLS (red dashed-line) to the Exact Data Points (blue line).

Second, the code implemented another method to predict the time-series, which is the iterative prediction method. The iterative prediction method starts by using the training data as an input data set to predict an output, as usual. Next,

[‡]Following resources were used to implement the codes:
http://web.mit.edu/wingated/www/resources.html
https://gist.github.com/caub/9462102

the method augments the previously predicted output to the input data set and predicts another output, and this process keeps repeating to generate multiple outputs that make up the predicted time-series. The iterative prediction method predicted the time-series and obtained an NMSE of 0.042877 shown in Figure 7 above. Although the obtained NMSE improved over NMSE acquired by the first method, it is still worst than the NMSE achieved by the authors of [4] and the competition winner. Although the three NMSEs were obtained by the same algorithm, different values of the NMSE were obtained. This is clearly caused by the approach used to predict the time-series.

4 KAARMA Algorithm

4.1 Learning Algorithm

Previously, Section 2 derived, explained, and analyzed the time-series model known as the autoregressive moving average (ARMA) model. In this case, a kernel function is utilized and the ARMA model is trained in the Hilbert space $\mathcal{H}$ to produce a kernel adaptive filtering (KAF) algorithm called the kernel adaptive autoregressive moving average (KAARMA) algorithm. In this section of the paper, the nonlinear version of the ARMA model, namely the kernel adaptive ARMA (KAARMA) algorithm, will be addressed.

As in the case of the kernel methods, solutions of a kernel adaptive filtering (KAF) algorithm can be expressed in equation (55). Compared with the previously addressed algorithms that are composed of inputs and outputs, the KAARMA algorithm has an additional state vector. Furthermore, the authors of [2] proposed a gradient-descent adaptive procedure that uses an input sequence and the observed outputs to learn the unknowns of a nonlinear system.

To demonstrate the Kernel Adaptive ARMA (KAARMA) algorithm mathematically, a dynamical system should be introduced by the following equations

$$\mathbf{x}_i = \mathbf{f}(\mathbf{x_{i-1}}, \mathbf{u_i}) \tag{60}$$

$$\mathbf{y}_i = \mathbf{h}(\mathbf{x_i}) \tag{61}$$

where $\mathbf{f}(\cdot, \cdot)$ is a continuous nonlinear state function and $\mathbf{h}(\cdot)$ is a continuous

nonlinear output function, and they can be expressed as follows

$$\begin{aligned}\mathbf{f}(\mathbf{x_{i-1}}, \mathbf{u_i}) &\triangleq \left[f^{(1)}(\mathbf{x}_{i-1}, \mathbf{u}_i),\ \ldots,\ f^{(n_x)}(\mathbf{x}_{i-1}, \mathbf{u}_i)\right]^{\mathrm{T}} \\ &= \left[\mathbf{x}_i^{(1)}, \ldots, \mathbf{x}_i^{(n_x)}\right]^{\mathrm{T}}\end{aligned} \tag{62}$$

$$\begin{aligned}\mathbf{h}(\mathbf{x}_i) &\triangleq \left[h^{(1)}(\mathbf{x}_i),\ \ldots,\ h^{(n_y)}(\mathbf{x}_i)\right]^{\mathrm{T}} \\ &= \left[\mathbf{y}_i^{(1)}, \ldots, \mathbf{y}_i^{(n_y)}\right]^{\mathrm{T}}\end{aligned} \tag{63}$$

in which $\mathbf{u}_i \in \mathbb{R}^{n_u}$ represents the input vector, $\mathbf{x}_i \in \mathbb{R}^{n_x}$ represents the state vector, $\mathbf{y}_i \in \mathbb{R}^{n_y}$ represents the output vector, $(n_u,\ n_x,\ n_y)$ represents the dimensions for each vector, and the subscript within brackets $^{(k)}$ represents the k-th component of a vector.

In addition, the authors of [2] presented a "new hidden state" vector that can be expressed using equation (60) and equation (61) as shown in the following equation

$$\mathbf{s}_i \triangleq \begin{bmatrix}\mathbf{x}_i \\ \mathbf{y}_i\end{bmatrix} = \begin{bmatrix}\mathbf{f}(\mathbf{x_{i-1}}, \mathbf{u_i}) \\ \mathbf{h}(\mathbf{f}(\mathbf{x_{i-1}}, \mathbf{u_i}))\end{bmatrix} \tag{64}$$

Using the expression previously stated in equation (64), y_i can be reformulated to the equation below

$$y_i = \underbrace{\begin{bmatrix}\mathbf{0} & \mathbf{I}_{n_y}\end{bmatrix}}_{\mathbb{I}} \underbrace{\begin{bmatrix}x_i \\ y_i\end{bmatrix}}_{\mathbf{s}_i} = \mathbb{I}\mathbf{s}_i \tag{65}$$

where $\mathbb{I}$ is the fixed selector matrix, $\mathbf{I}_{n_y}$ is a $n_y \times n_y$ identity matrix, $\mathbf{0}$ is an $n_y \times n_x$ zero matrix, and $\mathbf{s}_i \in \mathbb{R}^{n_s}$.

Moreover, to allow the KAARMA algorithm to learn the unknowns of a nonlinear system, specifically the continuous state function $\mathbf{f}(\cdot, \cdot)$ and the output function $\mathbf{h}(\mathbf{f}(\cdot, \cdot))$, the authors of [2] applied the theory of the reproducing kernel Hilbert space $\mathcal{H}$. For instance, to apply the theory of RKHS to the system in equation (60) and equation (61), the inputs $\mathbf{x}_{i-1}$ and $\mathbf{u}_i$ will be mapped separately into the Hilbert space $\mathcal{H}$ reproducing $\varphi(\mathbf{x}_{i-1}) \in \mathcal{H}_x$ and $\phi(\mathbf{u}_i) \in \mathcal{H}_u$, respectively. Next, the tensor product of the feature vectors will be taken to produce the tensor-product feature expressed in the following equation

$$\psi(\mathbf{x}_{i-1}, \mathbf{u}_i) \triangleq \varphi(\mathbf{x}_{i-1}) \otimes \phi(\mathbf{u}_i) \in \mathcal{H}_{su} \tag{66}$$

Furthermore, the authors [2] did not provide an analytical example of "how to" map the input vectors into the Hilbert space $\mathcal{H}$. However, the authors of [7] provided a concrete example that shows how the input vectors can be mapped into the Hilbert space $\mathcal{H}$ resulting in a feature vector. However, the kernel used in their example was the polynomial kernel.

The authors of [2] also introduced the weights $\mathbf{\Omega}$ in which the equation of the weights $\mathbf{\Omega}_i$ at time i can be expressed as the linear combination of the previous features as stated in the following equation

$$\mathbf{\Omega}_i = \mathbf{\Psi}_i \mathbf{A}_i \tag{67}$$

where

$$\mathbf{\Psi}_i \triangleq \big[\psi(\mathbf{x}_{-1}, \mathbf{u}_0),\ \ldots,\ \psi(\mathbf{x}_{m-2}, \mathbf{u}_{m-1})\big] \in \mathbb{R}^{n_\psi \times m}$$

is a vector of the m previous tensor-product features, and

$$\mathbf{A}_i \triangleq \big[\alpha_{i,1},\ \ldots,\ \alpha_{i,n_s}\big] \in \mathbb{R}^{m \times n_s}$$

are the corresponding coefficients, and m is the dictionary size in which $\mathbf{\Psi}_i$ represents the dictionary also.

Additionally, the authors of [2] introduced the cost function ϵ_i at time i, and it can be expressed as the following equation

$$\epsilon_i = \frac{1}{2} \mathbf{e}_i^{\mathrm{T}} \mathbf{e}_i \tag{68}$$

where $\mathbf{e}_i = \mathbf{d}_i - \mathbf{y}_i \in \mathbb{R}^{n_y \times 1}$ is the error vector with $\mathbf{d}_i$ representing the desired output.

By using the equation of the cost function expressed in equation (68), the error gradient with respect to $\mathbf{\Omega}_i$ can be written as the following equation

$$\frac{\partial \epsilon_i}{\partial \mathbf{\Omega}_i} = \frac{\partial \mathbf{e}_i^{\mathrm{T}} \mathbf{e}_i}{2 \partial \mathbf{\Omega}_i} = \mathbf{e}_i^{\mathrm{T}} \frac{\partial \mathbf{y}_i}{\partial \mathbf{\Omega}_i} \tag{69}$$

4.2 Comparison with KRLS

As the previous section of the paper showed, the KAARMA and the KRLS algorithms share some properties; however, the algorithms have various differences between them. This section of the paper will go through three major differences that can be observed when comparing the two algorithms.

As the authors of [2] stated, the state vector used in the KRLS algorithm can be expressed by

$$\mathbf{x}_i = \lambda^{-1/2}\mathbf{x}_{i-1}$$

such that $0 < \lambda \leq 1$ is a constant forgetting factor. On the other hand, as it was expressed in equation (60), the state vector utilized in the KAARMA algorithm can be written as

$$\mathbf{x}_i = \mathbf{f}(\mathbf{x_{i-1}}, \mathbf{u_i})$$

where $\mathbf{u}_i$ is the input vector and $\mathbf{x}_{i-1}$ is the previous state vector.

Both, the KRLS and the KAARMA algorithms include a dictionary with size m. However, the dictionary $\mathcal{D}$ of the KRLS algorithm works in a different way than the dictionary $\mathbf{\Psi}_i$ of the KAARMA algorithm. For instance, by observing the pseudo-codes below, it can be realized that the KRLS algorithm initializes $m = 1$, and then the ALD test decides whether m should be increased or not. If the ALD condition was not satisfied, a training sample x_t will be added to the dictionary, and m increases by 1. However, if ALD condition was not satisfied, then the dictionary size will not change. On the other hand, the KAARMA algorithm initializes $m = 1$: dictionary size; in other words, m can be any arbitrary value. Furthermore, m represents the numbers of past tensor-product features contained in Ψ_i.

When comparing the two algorithms with respect to the process of mapping the inputs into the Hilbert space $\mathcal{H}$, it can be observed that each algorithm has a different mapping process that produces a feature vector or a feature matrix. Both algorithms map the input vectors to the Hilbert space $\mathcal{H}$ producing a feature vector $\varphi(\cdot)$. However, in the case of the KRLS algorithm, producing a feature vector means that the mapping process is done and the ALD test can carry on to check if the feature vectors satisfy the ALD condition or not, as shown in the pseudo-code of algorithm (1). On the other hand, in the case of the KAARMA algorithm, after mapping the two inputs into two separate Hilbert spaces $\mathcal{H}$ producing two feature vectors $\varphi(\cdot)$ and $\phi(\cdot)$, a tensor product of the two feature vectors is taken to produce a tensor-product feature $\psi(\cdot,\cdot) = \varphi(\cdot) \otimes \phi(\cdot)$ that makes up the feature matrix $\mathbf{\Psi}$, as shown in the pseudo-code of algorithm (2).

Algorithm 1 Kernel RLS Algorithm

Parameter: ν
Initialization:
$\tilde{\mathbf{K}}_1 = [k_{11}]$
$\tilde{\mathbf{K}}_1^{-1} = \left[\frac{1}{k_{11}}\right]$
$\tilde{\alpha}_1 = \left(\frac{y_1}{k_{11}}\right)$
$\mathbf{P}_1 = [1]$
$m = 1$
for $t = 2, 3, \ldots$ **do**
  1. Get new sample: (x_t, y_t)
  2. Compute $\tilde{\mathbf{k}}_{t-1}(\mathbf{x}_t)$
  3. ALD test:
  $\boldsymbol{a}_t = \tilde{\mathbf{K}}_{t-1}\tilde{\mathbf{k}}_{t-1}(\mathbf{x}_t)$
  $\delta_t = k_{tt} - \tilde{\mathbf{k}}_{t-1}(\mathbf{x}_t)^{\mathrm{T}}\boldsymbol{a}_t$
  if $\delta_t > \nu$ **then** % add $\mathbf{x}_t$ to dictionary
    $\mathcal{D}_t = \mathcal{D}_{t-1} \cup \{\tilde{\mathbf{x}}_\mathbf{t}\}$
    Compute $\tilde{\mathbf{K}}_{t-1}$
    Compute $\mathbf{P}_t$
    Compute $\tilde{\alpha}_t$
    $m := m + 1$
  else % dictionary unchanged
    $\mathcal{D}_t = \mathcal{D}_{t-1}$
    $\mathbf{q}_t = \frac{\mathbf{P}_{t-1}\boldsymbol{a}_t}{1+\boldsymbol{a}_t^{\mathrm{T}}\mathbf{P}_{t-1}\boldsymbol{a}_t}$
    Compute $\mathbf{P}_t$
    Compute $\tilde{\alpha}_t$
  end if
end for
Output: $\mathcal{D}_t,\ \tilde{\alpha}_t$

Algorithm 2 Kernel Adaptive ARMA Algorithm

Initialization:
n_u: input dimension
n_s: state dimension
n_y: output dimension
n_s: state kernel parameter
n_u: input kernel paramter
η: learning rate
Randomly initialize input $\mathbf{u}_0 \in \mathbb{R}^{1\times n_u}$
Randomly initialize states $\mathbf{s}_{-1}$ and $\mathbf{s}_0 \in \mathbb{R}^{1\times n_s}$
Randomly initialize coefficient matrix $\mathbf{A} \in \mathbb{R}^{1\times n_s}$
$\boldsymbol{\Psi} = [\psi(\mathbf{s}_{-1}, \mathbf{u}_0]$: feature matrix
$\mathbf{S} = [\mathbf{s}_{-1}]$: state dictionary
m=1:dictionary size
$\mathbb{I} = [\mathbf{0} \quad \mathbf{I}_{n_y}] \in \mathbb{R}^{n_y\times n_s}$: measurement matrix
Computation:
for time $t = 1, \ldots, n$ **do**
  Initialization
  $\boldsymbol{\Psi}' = [\,]$: feature matrix update
  $\mathbf{S}' = [\,]$: state matrix update
  $\boldsymbol{V}_1^{(k)} = \mathbf{I}_{n_s}^{(k)} \in \mathbb{R}^{n_s\times 1}$, for $k = 1, \ldots, n_s$
  Update State-Transition Gradient Matrix
  for time $i = 1, \ldots, t$ **do**
    Generate Next State
    $\mathbf{s}_i = \boldsymbol{\Omega}^{\mathrm{T}}\psi(\mathbf{s}_{-1}, \mathbf{u}_i)$
    Update State Gradient
    $\mathbf{D}_i = [(\mathbf{S}^{(1)} - \mathbf{s}_{i-1}), \ldots, (\mathbf{S}^{(m)} - \mathbf{s}_{i-1})]$
    $\mathbf{K}_i = diag(\boldsymbol{\Psi}^{\mathrm{T}}\psi(\mathbf{s}_{i-1}, \mathbf{u}_i))$
    $\boldsymbol{\Lambda}_i = 2a_s\mathbf{A}^{\mathrm{T}}\mathbf{K}_i\mathbf{D}_i^{\mathrm{T}}$
    $\boldsymbol{V}_i^{(k)} = [\boldsymbol{\Lambda}_i\boldsymbol{V}_{i-1}^{(k)},\ \mathbf{I}_{n_s}^k]$
    Update Feature and State Matrices
    $\boldsymbol{\Psi}' = [\boldsymbol{\Psi}',\ \psi(\mathbf{s}_{i-1}, \mathbf{u}_i)]$
    $\mathbf{S}' = [\mathbf{S}',\ \mathbf{s}_{i-1}]$
  end for
  Prediction
  $\mathbf{y}_t = \mathbb{I}\mathbf{s}_t$
  Update Weights in the RKHS
  $\mathbf{e}_t = \mathbf{d}_t - \mathbf{y}_t$
  $\mathbf{A}^{(\mathrm{k})} = \begin{bmatrix} \mathbf{A}_i^{(\mathrm{k})} \\ \eta\left(\mathbb{I}\boldsymbol{V}_i^{(k)}\right)^{\mathrm{T}}\mathbf{e}_i \end{bmatrix}$
  $\boldsymbol{\Psi} = [\boldsymbol{\Psi}, \boldsymbol{\Psi}']$
  $\mathbf{S} = [\mathbf{S},\ \mathbf{S}']$
  $\boldsymbol{\Omega} = \boldsymbol{\Psi}\mathbf{A}$
  $m = m + t$
end for

5 Conclusion

To sum up, this paper presented a linear time series model called the autoregressive moving average (ARMA) model. Also, the paper derived a learning algorithm for the linear model and showed that it is solvable by estimating the unknown parameters using two methods, namely the least squares method and the recursive least squares (RLS) method. The paper also showed how a kernel function can be used to produce a kernel method as the kernel recursive least squares (KRLS) algorithm, which is the linear version of the RLS. The KRLS algorithm was used for time-series prediction (TSP), and the obtained results were compared with previously reported results. At the end, the showed how a kernel function can be utilized to produce a kernel adaptive filtering algorithm as the kernel adaptive autoregressive moving average (KAARMA) algorithm. The paper also compared between the KRLS algorithm and the KAARMA algorithm as it pointed to few major differences between the two algorithms.

6 References

[1] V. N. Vapnik, *The nature of statistical learning theory.* Springer-Verlag New York, Inc., 1995.

[2] K. Li and J. C. Príncipe, "The kernel adaptive autoregressive-moving-average algorithm," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 27, no. 2, pp. 334–346, 2016.

[3] W. Y. Yang, W. Cao, T.-S. Chung, and J. Morris, *System of Linear Equations.* John Wiley & Sons, Ltd, 2005, ch. 2, pp. 71–115. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/0471705195.ch2

[4] Y. Engel, S. Mannor, and R. Meir, "The kernel recursive least-squares algorithm," *IEEE Transactions on Signal Processing*, vol. 52, no. 8, pp. 2275–2285, 2004.

[5] A. Weigend and N. Gerschenfeld, *Time Series Prediction: Forecasting the Future and Understanding the Past*, 06 1994, vol. 10.

[6] A. Weigend and N. Gershenfeld, "Results of the time series prediction competition at the santa fe institute," in *IEEE International Conference on Neural Networks*, 1993, pp. 1786–1793 vol.3.

[7] J. C. Príncipe, W. Liu, and S. Haykin, *Kernel adaptive filtering: a comprehensive introduction.* John Wiley & Sons, 2011.

7 Appendix [§]

7.1 MATLAB Code for the Learning Algorithm

```
clc; clear; clf; close all;

%% LearningAlgorithmCode.m Explanation:

% Is the m file name suggest. This is a code that I wrote to implement the
% learning algorithm explained in the research paper. The code includes two
% phases. A phase of simulating data, and then a phase of estimating the
% system parameters f,g,h.

%% Generating Data Phase:

% To Control Random Number Generator
rng(10)

iter = 100;

x = zeros(iter,1);
y = zeros(iter,1);
s = zeros(2, iter);

x(1) = 2;
y(1) = 1;
s(:,1) = [x(1) ; y(1)];

f_exact = 0.5;
g_exact = 1;
h_exact = 2;

```

[§]All MATLAB codes are available in the following Github repository: https://github.com/7mxd/Senior-Research-Project-2022-2023

```
% Using u as a random response instead of the
    commented impulse response

u = zeros(iter, 1);
u = randi([0,1], [1,iter]);

% for i = 1:iter
%     if i == 1
%         %u(i) = 1;
%     else
%         %u(i) = 0;
%     end
% end

d = zeros(iter, 1);

for i = 2:iter
    x(i) = f_exact*x(i-1) + g_exact*u(i);
    y(i) = h_exact*x(i);
    s(:, i) = [x(i) ; y(i)];
end

for i = 1:iter
    d(i) = y(i);
end

%% Estimating the System Parameters:
% Note: I used the explicit scheme.

n = 0.1; % learning rate
f = zeros(iter, 1);
g = zeros(iter, 1);
h = zeros(iter, 1);
f(1) = 0.3;
g(1) = 0.7;
h(1) = 1;
```

```
64
65
66  e = zeros(iter, 1);
67  yhat = zeros(iter, 1);
68  yhat(1) = 1;
69
70
71  for i = 2:iter % # of iterations
72      f(i) = f(i-1) + n*e(i-1)*yhat(i-1); % yhat(i-1) =
            I*s(:, i-1)
73      g(i) = g(i-1) + n*e(i-1)*h(i-1)*u(i);
74      h(i) = h(i-1) + n*e(i-1)*(yhat(i-1)/h(i-1));
75      yhat(i) = f(i)*yhat(i-1) + h(i)*g(i)*u(i);
76      e(i) = d(i) - yhat(i);
77  end
78
79  %% Plotting and Calculating Errors
80
81  Error_f = zeros(iter, 1);
82  Error_g = zeros(iter, 1);
83  Error_h = zeros(iter, 1);
84
85  for i = 1:iter
86      Error_f(i) = ((abs(f(i) - f_exact))/f_exact)*100;
87      Error_g(i) = ((abs(g(i) - g_exact))/g_exact)*100;
88      Error_h(i) = ((abs(h(i) - h_exact))/h_exact)*100;
89  end
90
91  Error = [Error_f, Error_g, Error_h];
92
93  figure('Name', 'Estimates of f,g,h')
94  % Figure 1 (Estimates of f,g,h for 100 Iterations)
95
96  subplot(3,1,1)
97  plot([1:100]', f)
98  yline(0.5, '--r', 'f')
```

```
axis([-inf inf 0 2.5])
xlabel('Number of Iterations')
ylabel('f Value')
title('Estimates of f for 100 Iterations')

subplot(3,1,2)
plot([1:100]', g)
yline(1, '--r', 'g')
axis([-inf inf 0 2.5])
xlabel('Number of Iterations')
ylabel('g Value')
title('Estimates of g for 100 Iterations')

subplot(3,1,3)
plot([1:100]', h)
yline(2, '--r', 'h')
axis([-inf inf 0 2.5])
xlabel('Number of Iterations')
ylabel('h Value')
title('Estimates of h for 100 Iterations')

table([1:100]', f, Error_f, g, Error_g, h, Error_h, '
    VariableNames', ...
     {'iteration', 'f est.', 'f Error%', 'g est.', 'g
        Error%', 'h est.', ...
     'h Error%'})

% To export .eps figure
% print -depsc fghestimatesLA
```

7.2 MATLAB Codes for the Least Squares Method

7.2.1 Least Squares Method Function

```
%% Algorithm_LSE.m Explanation:

```

```
% A function that takes two inputs and returns three outputs

% Inputs:
% - iter : Number of Iterations for Generating Data
% - rows : Number of Data Points Used for Estimating x

% Note than x is called here as (z) to avoid some confusions when running
% the code

% Outputs:
% Az = b ; (x is replaced by z)
% Outputs are A matrix, z vector, and b vector.

%% Code Implementation

function [z,b,A] = Algorithm_LSE(iter, rows)


% Generating Data Phase:
x = zeros(iter,1);
y = zeros(iter,1);
s = zeros(2, iter);

x(1) = 2;
y(1) = 1;
s(:,1) = [x(1) ; y(1)];

f = 0.5;
g = 1;
h = 2;

u = zeros(iter, 1);
u = randi([0,1], [iter,1]);
% for i = 1:iter
```

```
%     if i == 1
%         %u(i) = 1;
%     else
%         %u(i) = 0;
%     end
% end

d = zeros(iter, 1);

for i = 2:iter
    x(i) = f*x(i-1) + g*u(i);
    y(i) = h*x(i);
    s(:, i) = [x(i) ; y(i)];
end

for i = 1:iter
    d(i) = y(i);
end

% Solving  b = Az


% for i=1:iter-1
%     b(i,1) = y(i+1);
% end
%
% for i=1:iter-1
%     A(i, 1) = y(i);
%     A(i, 2) = u(i+1);
% end

for i=1:rows
    b(i,1) = y(i+1);
end

for i=1:rows
```

```
    A(i,1) = y(i);
    A(i,2) = u(i+1);
end

z = A\b;
end
```

7.2.2 Least Squares Method Estimations Analysis

```
clc; clear; clf;

% We have the following:
                              f = 0.5;
                              g = 1;
                              h = 2;
% Where beta is f*g as follows:
                              Beta = g*h;

% We are going through 100 iteration to generate the
    data

% Therefore, to check the impact of data size, I will
    keep increasing the
% number of rows to estimate f and B.

% I will test the following sizes
                            datasize = [25, 50, 75,
                               100];

% Set seed to get fixed results:
rng(1);

[z1,b1,A1] = Algorithm_LSE(101, 25);
ERR_PERC(1,1) = ((abs(z1(1) - f))/f)*100;
ERR_PERC(1,2) = ((abs(z1(2) - Beta))/Beta)*100;

[z2,b2,A2] = Algorithm_LSE(101, 50);
```

```
ERR_PERC(2,1) = ((abs(z2(1) - f))/f)*100;
ERR_PERC(2,2) = ((abs(z2(2) - Beta))/Beta)*100;

[z3,b3,A3] = Algorithm_LSE(101, 75);
ERR_PERC(3,1) = ((abs(z3(1) - f))/f)*100;
ERR_PERC(3,2) = ((abs(z3(2) - Beta))/Beta)*100;

[z4,b4,A4] = Algorithm_LSE(101, 100);
ERR_PERC(4,1) = ((abs(z4(1) - f))/f)*100;
ERR_PERC(4,2) = ((abs(z4(2) - Beta))/Beta)*100;

f_values = [z1(1), z2(1), z3(1), z4(1)];
beta_values = [z1(2), z2(2), z3(2), z4(2)];

table(datasize', f_values', ERR_PERC(:,1), beta_values', ERR_PERC(:,2), ...
    'VariableNames', {'Data Points', 'f value', 'f Error %', 'beta value', 'beta Error %'})


subplot(5,1,[1,2])
plot(datasize, ERR_PERC(:, 1), '-o')
xticks(datasize)
ytickformat('percentage')
axis([-inf inf -inf inf])
xlabel("Data Points")
ylabel("Error Perctenage")
for i=1:numel(datasize)
    text(datasize(i) + 1.2, ERR_PERC(i,1), [num2str(ERR_PERC(i,1), '%0.2f'),'%'])
end
title("Error Percentage Compared with Data Size for f", 'fontName', 'Times New Roman')

subplot(5,1,[4,5])
plot(datasize, ERR_PERC(:, 2), 'r-o')
```

```
xticks(datasize)
ytickformat('percentage')
axis([-inf inf -inf inf])
xlabel("Data Points")
ylabel("Error Perctenage")
for i=1:numel(datasize)
    text(datasize(i) + 1.2, ERR_PERC(i,2), [num2str(
        ERR_PERC(i,2), '%0.2f'),'%'])
end
title("Error Percentage Compared with Data Size for \
    beta",'fontName', 'Times New Roman')

% To export .eps figure
print -depsc LSEstimations
```

7.3 MATLAB Codes for the RLS Method

7.3.1 RLSE_Online.m Function

```
%% RLSE on-line processing
%  Implemeneted by following instructions given by (
    APPLIED NUMERICAL
%  METHODS USING MATLAB), pages 76-79

%% Explanation:

% at_k1 represents the added data point of A matrix
% b_k1 represents the added data point of b vector
% K represents the gain matrix
% P represents the inverse matrix "[A_k' A_k]^-1"

%% Defining RLSE_Online Function:

function [x, K, P] = RLSE_Online(aT_k1, b_k1, x, P)
    K = P*aT_k1' / (aT_k1*P*aT_k1' + 1); % Equation
        (2.1.17) Page 78
```

```
    x = x + K * (b_k1 - aT_k1*x); % Equation (2.1.16) Page 78
    P = P - K*aT_k1*P; % Equation (2.1.18) Page 78
end
```

7.3.2 RLS Algorithm Analysis

```
%% Algorithm_RLSE Analysis:

clear; clc; clf; close all;

rng(1);

% Note that we have the following parameters:
                                    %f = 0.5;
                                     Beta = 2;

% and I will test the following numbers of data points;
                                k_data = [26, 51, 76, 101];

%% Program Implementation:

% Initializing Data:
iter = 101; % Number of Iterations
x = zeros(iter,1); % `x` in learning algorithm
y = zeros(iter,1); % `y` in learning algorithm
x(1) = 2;
y(1) = 1;
u = randi([0,1], [iter, 1]);
f = 0.5;
g = 1;
h = 2;

z_o = [0.5 2]'; % True Values of the Parameters (f, beta)
```

```
zsize = length(z_o);
z = zeros(zsize, 1);
P = 100*eye(zsize, zsize);

for k=1:(iter - 1)
    x(k+1) = f*x(k) + g*u(k+1);
    y(k+1) = h*x(k+1);

    A(k, :) = [y(k) u(k+1)];
    b(k, :) = y(k+1);
    [z,K,P] = RLSE_Online(A(k,:), b(k,:), z, P); % Updating z vector (f, beta) estimates
    online(:,k) = z;

    ERR_PERC(k,1) = ((abs(online(1,k) - f))/f)*100;
    ERR_PERC(k,2) = ((abs(online(2,k) - Beta))/Beta)*100;
end

%% Table & Plots:

table1 = table([2:101]', online(1,:)', ERR_PERC(:,1), online(2,:)', ERR_PERC(:,2), ...
    'VariableNames', {'k', 'f value', 'f Error %', 'beta value', 'beta Error %'})

figure('Name', 'Estimates of f and beta for different k values')
% Figure 1 (Estimates of f and beta for different data points)
subplot(2,1,1)
plot([2:101]', online(1,:)')
yline(0.5, '--r', 'f')
xlim([2 101])
xticks([2:11:102])
ylim([0 2.5])
```

```
xlabel('k value')
ylabel('f value')
title('f estimates of x_{k} s.t. k\in[2, 101]','
    fontName', 'Times New Roman')

subplot(2,1,2)
plot([2:101]', online(2,:))
yline(2, '--r', '\beta')
xlim([2 101])
xticks([2:11:102])
ylim([1.5 inf])
xlabel('k value')
ylabel('\beta value')
title('\beta estimates of x_{k} s.t. k\in[2, 101]','
    fontName', 'Times New Roman')

% To export .eps figure
print -depsc RLSEstimations1

figure('Name', "Error Percentage Compared with k
    values for f and beta")
% Figure 2 (Error Percentage Compared with Data Size
    for f and beta)
ERR = [ERR_PERC(25,:)', ERR_PERC(50,:)', ERR_PERC
    (75,:)', ERR_PERC(100,:)'];
subplot(5,1,[1,2])
plot(k_data, ERR(1, :), '-o')
xticks(k_data)
ytickformat('percentage')
axis([-inf inf -inf inf])
xlabel("k value")
ylabel("Error Perctenage")
for i=1:numel(k_data)
    text(k_data(i) + 1.2, ERR(1,i), [num2str(ERR(1,i),
        '%0.2f'),'%'])
end
```

```
title("Error Percentage for f Estimates at Different k
    values of x_{k}",'fontName', 'Times New Roman')

subplot(5,1,[4,5])
plot(k_data, ERR(2, :), 'r-o')
xticks(k_data)
ytickformat('percentage')
axis([-inf inf -inf inf])
xlabel("k value")
ylabel("Error Perctenage")
for i=1:numel(k_data)
    text(k_data(i) + 1.2, ERR(2,i), [num2str(ERR(2,i),
        '%0.2f'),'%'])
end
title("Error Percentage for \beta Estimates at
   Different k values of x_{k}",'fontName', 'Times New
    Roman')

% To export .eps figure
print -depsc RLSEstimations2
```

7.4 MATLAB Codes for the KRLS Algorithm

7.4.1 KRLS MATLAB Functions

```
%% rbf4nn Function Explanation:

% Inputs:
% X -> a matrix that contains all samples as columns
% Y -> another matrix that contains all samples as
   columns
% sigsq -> sigma (squared) ; where sigma represents
   the kernal width

% Outputs:
% K_ij = exp(-1 / (2*sigsq) * (X_i' * Y_j))
% where K_ij is the Gaussian kernel function equation
```

```
11
12 function K = rbf4nn(X, Y, sigsq)
13 K = X' * Y;
14
15 Xsq = X .* X;
16 Xsum = sum(Xsq, 1);
17
18 Ysq = Y .* Y;
19 Ysum = sum(Ysq, 1);
20
21 K = K - Xsum' * ones(1, length(Ysum)) / 2;
22 K = K - ones(length(Xsum), 1) * Ysum / 2;
23 K = K ./ sigsq;
24 K = exp(K);
25 end
```

```
1  %% dict_init Function Explanation:
2
3  % Inputs:
4  % kfunc -> Handle to the kernel function.
5  % kparam -> The "kernel specific" parameter that will
       be passed to the kernel function on every call.
6  % thresh -> the almost-linearly-independent threshold
7  % state -> the initial data point (column vector)
8
9  % Outputs:
10 % dp -> A dictionary data structure. Used to be called
       in "dict.m".
11
12 function dp = dict_init(kfunc, kparam, thresh, state)
13 dp.kfunc = kfunc;
14 dp.kparam = kparam;
15 dp.thresh = thresh;
16
17 dp.Dict = state;
18
19 dp.K = feval(kfunc, state, state, kparam); % evaluate
```

```
    the kernel function on the three other inputs
dp.Kinv = 1 / dp.K; % inverted K

dp.addedFlag = 1;

return;
end
```

```
%% dict Function Explanation:

% Inputs:
% dp -> Data structure represting the problem setup, returned from
% dict_init.m or subsequent calls to this function (dict.m)
% state -> A new data point

% Outputs:
% dp -> An updated data structure.

function dp = dict(dp, state)
m = size(dp.Dict, 2); % Number of Enteries ; note that size('',2) returns the columns number

ktt = feval(dp.kfunc, state, state, dp.kparam);
ktwid = feval(dp.kfunc, dp.Dict, state, dp.kparam);

at = dp.Kinv * ktwid;
dt = ktt - ktwid' * at;

dp.addedFlag = 0;

if (dt > dp.thresh) % if NOT almost-linearly-dependent, add to dictionary
    dp.Dict = [dp.Dict, state];
    dp.K = [dp.K, ktwid; ktwid', ktt];
    dp.Kinv = (1 / dt) * [dt * (dp.Kinv) + at * at', -
```

```
        at; -at', 1];
    dp.addedFlag = 1;
end

dp.at = at;
dp.dt = dt;
dp.ktwid = ktwid;
kp.ktt = ktt;

return;
end
```

```
%% krls_init Function Explanation:

% Inputs:
% kfunc -> Handle to the kernel function.
% kparam -> The "kernel specific" parameter that will
    be passed to the kernel function on every call.
% thresh -> the almost-linearly-independent threshold
% state -> the initial data point (column vector)
% target -> the initial target (scalar)

% Outputs:
% kp -> A data structure that includes everything
    needed for subsequent KRLS calls (in krls.m)

function kp = krls_init(kfunc, kparam, thresh, state,
    target)
kp.kfunc = kfunc;
kp.kparam = kparam;
kp.thresh = thresh;

kp.dp = dict_init(kfunc, kparam, thresh, state);

kp.P = 1;
kp.Alpha = target / kp.dp.K;

```

```
return;
end
```

```
%% krls Function Explanation:

% Inputs:
% kp -> the data structure returned from "krls_init.m" or subsequent calls to this function (krls.m)
% state -> the next state vector (data point) (column vector)
% target -> the target value (scalar)

% Outputs:
% kp -> A data structure that includes everything needed for subsequent KRLS calls (in krls.m)

function kp = krls(kp, state, target)
m = size(kp.dp.Dict, 2); % Number of Enteries ; note that size('',2) returns the columns number

kp.dp = dict(kp.dp, state);

at = kp.dp.at;
dt = kp.dp.dt;
ktwid = kp.dp.ktwid;
Kinv = kp.dp.Kinv;

if(kp.dp.addedFlag)
    kp.P = [kp.P, zeros(m, 1); zeros(1, m), 1];
    inno = (target - ktwid' * kp.Alpha) / dt;
    kp.Alpha = [kp.Alpha - at * inno; inno];
    kp.addedFlag = 1;
else
    tmp = kp.P * at;
    qt = tmp / (1 + at' * tmp);
    kp.P = kp.P - qt * tmp';
    kp.Alpha = kp.Alpha + Kinv * qt * (target - ktwid'
```

```
        * kp.Alpha);
    kp.addedFlag = 0;
end

return;
end
```

```
%% krls_query Function Explanation:

% Inputs:
% kp -> the data structure returned from "krls_init.m"
    or subsequent calls to the "krls.m" function
% state -> the vector you want to query

% Outputs:
% val -> the value of regression function at state

function val = krls_query(kp, state)
kernvals = feval(kp.kfunc, state, kp.dp.Dict, kp.
   kparam);

val = kernvals * kp.Alpha;

return;
end
```

7.4.2 KRLS Santa Fe TSP

```
clc; clear; clf; close all;

%% krls_santafe.m Explanation:

% This code is written to use the previous KRLS
   function and apply them to "The Santa Fe Time
   Series Competition Data Set A: Laser generated data
   ". Based on 1000 data points of the laser time-
```

```
    series, KRLS will be used to predict the next 100
    data points.
6
7   %% Load Data and Prepare Inputs
8
9   tic; % Time Started for NSME
10
11  % Read Content from Webpage:
12  weboptions('Timeout', 15);
13  data = [webread('https://web.archive.org/web
        /20160427182805if_/http://www-psych.stanford.edu
        :80/~andreas/Time-Series/SantaFe/A.dat') ...
14       webread('https://web.archive.org/web
            /20160427182805if_/http://www-psych.stanford.
            edu:80/~andreas/Time-Series/SantaFe/A.cont')];
15  y = textscan(data, '%f'); % Read Formatted Data (%f)
    from a text file (data)
16  y = y{1}(1:1100) / 256; % y{1} to access cell (access
    data)
17  figure('Name', 'Santa Fe Laset Time Series Data')
18  plot(y(1:1000))
19
20
21  %% Set an Auto-Regressive Matrix for the inputs
22  X = zeros(length(y), 40); % 1100 rows and 40 columms
23  for i = 1:size(X, 2) % 1:40
24       X(:, i) = [NaN(i,1); y(1:end-i)]; % 'NaN' on the
            Upper Triangle of the Matrix
25  end
26
27  %% Kernel Recursive Least Squares Regression
28
29  % Parameters:
30  %   Here we test them directly. Normally we must
    allocate room for
```

```
%   cross-validation in order to find optimal
    parameters in a grid. Once
%   the best parameters are assessed, they an be
    evaluated on the
%   training + validation sets

kernel_func = @rbf4nn; % Gaussian Kernel
kparam = 0.9; % Variance of the Gaussian
ald_thresh = 0.01; % almost-linearly-dependance
    threshold
n = 40; % Auto-Regressive Window Size

ltest = 100; % Test Set Size
Ytest = y(end - ltest + 1 : end); % Test Set
X_ = X(n + 1 : end - ltest, 1:n); % 960 x 40
Y_ = y(n + 1 : end - ltest); % 960 x 1

% We could try to train with missing values, starting
    at 2 instead of n+1
ltraining = size(X_, 1); % size of training data (960)
lvalidation = 0;

kp = krls_init(kernel_func, kparam, ald_thresh, X_
    (1,:)', Y_(1));

for i  = 2:ltraining
    kp = krls(kp, X_(i,:)', Y_(i));
end

v = [Y_(ltraining) X_(ltraining, 1:n-1)];

prediction = zeros(ltest, 1);
for j = 1:ltest
    prediction(j) = krls_query(kp, v');
    v = [prediction(j) v(1:n-1)];
end
```

```
62
63 testNSME = goodnessOfFit(Ytest, prediction, 'NMSE');
64 disp("NSME = " + testNSME)
65
66 figure('Name', 'KRLS predicting 100 steps into the
       future with NMSE = 0.069428')
67 plot(1000 + (1:length(Ytest)), Ytest, 'b-')
68 hold on
69 plot(1000 + (1:length(Ytest)), prediction, 'r--')
70 hold off
71 legend('True Continuation', 'KRLS Prediction')
72
73 toc % Time Ended for NMSE
74
75 %% Reinforcement Training
76
77 % To improve the performance of a machine learning
       algorithm, a technique
78 % called "Reinforcement Training" is used to
       iteratively add new training
79 % samples into the existing dataset and re-train the
       model on the
80 % "augmented" dataset.
81
82 tic % Time Started for Reinforcement Training
83
84 ireinforce = 40; % We will have 10 values of NMSE
85 nmses = repmat(testNSME, 1, ireinforce); % Replicate
       the 'testNMSE' on 1 x 10 matrix
86 prediction = repmat(prediction, 1, ireinforce); %
       Replicate the 'prediction' on 1 x 10 matrix
87
88 for i = 2:ireinforce
89     % Make new inputs by injecting the fitted values
90     fitted = zeros(ltraining, 1); % fitted values are
           composed of 100 x 1 column vector
```

```
91      for j = i:ltraining
92          fitted(j) = krls_query(kp, X_(j, :)');
93      end
94      X_(2:end, 2:end) = X_(1:end-1, 1:end-1);
95      X_(i:end, 1) = fitted(1:end-i+1);
96      for j = i:ltraining
97          kp = krls(kp,  X_(j, :)', Y_(j));
98      end
99
100     v = [Y_(ltraining) X_(ltraining, 1:n-1)];
101     prediction = zeros(ltest, 1);
102     for j = 1:ltest
103         prediction(j) = krls_query(kp, v');
104         v = [prediction(j) v(1:n-1)];
105     end
106
107     nmses(i) = goodnessOfFit(Ytest, prediction, 'NMSE'
            );
108     predictions(:, i) = prediction; % Prediction
            Vector that will contain prediction vectors for
             each iteration
109 end
110
111 % Plot Best Reinforcement
112
113 [minNMSE, iNMSE]  = min(nmses); % 'iNMSE' is the index
        of the minimum NMSE
114 disp("MIN NSME = " + minNMSE)
115
116 figure('Name', 'KRLS predicting 100 steps into the
        future with NMSE = 0.042877')
117 plot(1000 + (1:length(Ytest)), Ytest, 'b-')
118 hold on
119 plot(1000 + (1:length(Ytest)), predictions(:, iNMSE),
        'r--')
120 hold off
```

```
121 legend('True Continuation', 'KRLS Prediction')
122
123 toc % Time Ended for Reinforcement Training
```