MSc – Adaptive Signal Processing Assignment

Jonathon Chambers, Danilo Mandic and Eric Grinstein

October 2021

0 Introduction

This assignment requires access to a computer running Python/Matlab. Furthermore, using Python's Jupyter Notebook or Matlab's Live Scripts is recommended as a way of writing code, plotting graphs and writing comments. Such notebooks may be exported as PDF documents and handed as this lab's report. Alternatively, the lab's report PDF file may also be generated using MS Word or Latex.

0.1 A Note on Notation

In this document, scalars are represented by lowercase, regular type-faced letters (example: x). Vectors are represented by lowercase boldface letters (example: $\boldsymbol{x}$). Matrices are represented by uppercase, boldface letters (example: $\boldsymbol{X}$)

0.2 Adaptive Algorithms for Echo Cancellation

Adaptive digital signal processing refers to the study of algorithms, architectures and learning strategies which have the capacity to vary in sympathy with changing statistical properties of the signal and environment. Such techniques have been successfully applied in many application areas, for example, channel equalisation in communications audio localization, Electrocardiogram (ECG) monitoring in medicine, as well as Electroencephalogram (EEG) conditioning for brain computer interfaces (BCIs).

This assignment is based upon the application of adaptive digital signal processing to echo cancellation. Namely, in e.g., online meeting scenarios, smart device which contains the loudspeaker and microphone is placed, for convenience, at some distance from the local participant. Therefore, when the remote (far-end) participant is talking, there is acoustic coupling between the loudspeaker and microphone of the mobile unit, which leads to the far-end participant hearing a disturbing echo of his/her own voice, a phenomenon routinely experienced in online teaching.

Elimination of this echo can be achieved with an adaptive filter, which models the time-varying acoustic coupling. The adaptive filter is applied in a system identification structure, as shown in Fig. 1. The input, $x(k)$, where k represents the sample index, corresponds to the loudspeaker signal, namely the far-end speech; the unknown system is the time-varying acoustic path between the loudspeaker and the microphone; the noise term, $n(k)$, represents that component of the microphone signal which has not been generated by the loudspeaker signal; and $d(k)$ is the microphone signal - the desired response for the adaptive filter. The adaptive filter attempts, in real-time, to track the time-variations of the acoustic path in order to generate, at its output, a signal, $y(k)$, which matches the component of the microphone signal due to the acoustic coupling between the loudspeaker and the microphone.

The adaptive filtering configuration is based upon a Finite Impulse Response (FIR) digital filter structure, together with an adaptation algorithm to adjust the coefficients of the filter in real time. The adaptation algorithm employs an optimization criterion, called a cost or an objective function, to adjust the coefficients of the FIR filter. The aim is to minimise some function of the error, $e(k) = d(k) - y(k)$, between the desired response, $d(k)$, and the output of the adaptive filter, $y(k)$. Typically, an estimate of error power $J = [e^2(k)]$ is employed in this context.

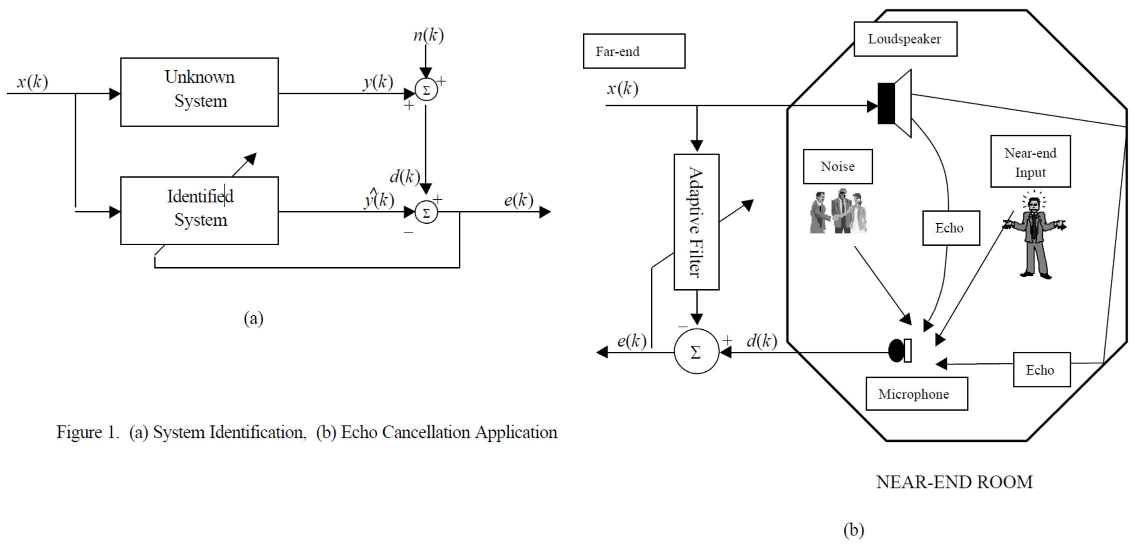

Figure 1: (a) System identification, (b) Echo cancellation application

0.3 Project

Two families of adaptation algorithms are to be investigated, namely the Least Mean Square (LMS) and Recursive Least Squares (RLS).

The basic LMS algorithm approximately minimises the mean square error $J = E[e^2(k)]$ where the error is given by $e(k) = d(k) - \boldsymbol{w}^T(k)\boldsymbol{x}(k)$ and $\boldsymbol{w}(k) = [w_1(k), w_2(k), ..., w_L(k)]^T$ is the coefficient vector of the adaptive filter of length L, and $\boldsymbol{x}(k) = [x(k), x(k-1), ..., x(k-L+1)]T$ is the adaptive filter data vector, that is the proportion of input data that are currently in the filter's memory. The update equation of the coefficient vector for the LMS algorithm is given by

$$\boldsymbol{w}(k+1) = \boldsymbol{w}(k) + 2\mu\boldsymbol{x}(k)e(k) \tag{1}$$

where the right hand term, $\boldsymbol{x}(k)e(k)$, is an instantaneous estimate of the negative gradient of the error performance surface J [Haykin, 1996], and μ is the adaptation gain, also called the step size. The coefficient vector of the adaptive filter is generally initialised to zero, that is, $\boldsymbol{w}(0) = \mathbf{0}$.

1 LMS

Write a function to implement the LMS algorithm. The function inputs should be vectors for the desired response signal, d(k), and the adaptive filter input, x(k), whereas scalars for the adaptation gain, μ, and the length of the adaptive filter L. The output should be the error signal, $e(k)$, together with the weight vector of the adaptive filter at each sample k.

The call of the function from within Python should have the form `errors, weights = lms(x, d, m, L)` where x and d are $N \times 1$ vectors, m and L are scalars and errors is a $N \times 1$ vector and weights is a $N \times L$ matrix.

Owing to the presence of feedback within the LMS algorithm, that is the output error is used to adjust the coefficients of the adaptive filter, may lead to instability. The adaptation gain, μ , is therefore the key parameter which controls how the adaptive filter behaves and it should be chosen to lie within the range [Mandic & Goh, 2009]

$$0 < \mu < 1/LP_x \tag{2}$$

where P_x is the power of the input signal to the adaptive filter.

Then, use the LMS routine implemented in a system identification setting. Represent the unknown system with a length $L = 9$ FIR filter with an optimal or desired impulse response sequence vector w_o of the form

$$\frac{1}{k} exp\big(- (k-4)^2/4\big) \tag{3}$$

where $k = 1, ..., 9$, and employ the `np.random.randn(200, 1)` in Python (or `randn(200, 1)` within MATLAB) to generate $x(k)$ which will have unit power and zero mean. The desired response signal is generated using the function `np.convolve(x, weights, mode="full")` in Python or `filter(w,[1],x)` in MATLAB.

1.1

Assume the additive noise, n(k), is zero and the length of the adaptive filter, L, equals that of the unknown system. Calculate the upper bound for the adaptation gain from equation (2), and multiply this by 1/3 [Bellanger, 1987].

Run the LMS algorithm with this setting for m and use plot() and semilogy() to observe the convergence of the squared error, $e^2(k)$, and the weight error vector norm with sample number - such plots give an indication of the learning rate of the adaptation algorithm. The normalised weight error vector norm is given by $\frac{||\boldsymbol{w}(k)-\boldsymbol{w}_o||^2}{||\boldsymbol{w}_o||^2}$

where $\boldsymbol{w}(k)$ is the coefficient vector of the adaptive filter at sample number k and the norm $||\boldsymbol{x}||^2 \cdot$ represents the sum of squared values of the vector argument $\boldsymbol{x}$. One example of Python code to achieve this is:

```
import numpy as np

def compute_error(w_k, w_o):
    numerator = np.sum((w_k - w_o)**2)
    denominator = np.sum(w_o**2)

    return numerator/denominator
```

Plot the final weight vector for the adaptive filter, $w(200)$.

Choose the length of the adaptive filter to be less than and greater than that of the unknown system. e.g. 5 and 11; recalculate the adaptation gain for each case, simulate and explain the results e.g. the final weight vectors and the effect upon the learning curves in terms of the undermodelling or overfitting.

1.2

Repeat 1.1 200 times for independent input sequences, i.e. call randn(200,1) and filter(w,[1],x) in MATLAB to generate new input and desired response vectors with the length of the adaptive filter equal to that of the unknown system. Plot the average squared error learning curve and the weight error vector norm misalignment - this corresponds to an approximate ensemble average [Haykin, 1996]. Comment on the results.

1.3

Repeat 1.2. for different settings of the adaptation gain μ e.g. [0.5,0.01,0.001]. What are the effects of increasing and decreasing the adaptation gain? Comment on the convergence and misadjustment of the LMS algorithm [Mandic & Goh, 2009].

1.4

Add zero mean, independent noise to the desired response signal so that the signal-to-noise ratio of the desired response is 20 dB, that is the ratio of the power of the output of the unknown system to the power of the independent noise is 100, and repeat 1.2. Comment on the convergence and misadjustment performance of the LMS algorithm.

2 NLMS

In the echo cancellation application, the input to the adaptive filter, is speech. A speech file called s1.mat is available which can be loaded with the *scipy.io.loadmat* command in Python, or the *load* command within MATLAB.

Use s1 as the input $\boldsymbol{x}(k)$. Comment upon the choice of adaptation gain for the LMS algorithm when the input to the adaptive filter is a real world speech signal.

A modification to the LMS algorithm is the Normalised LMS algorithm with an update equation of the form

$$\boldsymbol{w}(k+1) = \boldsymbol{w}(k) + \frac{2\mu}{1 + ||\boldsymbol{x}(k)||^2}\boldsymbol{x}(k)e(k) \tag{4}$$

Explain why this algorithm is more suitable for use with real world signals? Write a function to simulate this algorithm. Apply this algorithm to the system identification simulation with s1 as input. Comment upon its performance. Compare the computational complexities of the real-time implementations of the LMS and NLMS algorithms. What is the implication of the use of a fixed point precision digital signal processor chip upon the operation of a LMS adaptive filter [Haykin, 1996]?

3 RLS

In contrast to the stochastic cost function of the LMS, $J = E[e^2(k)]$, the RLS algorithm is based on the deterministict cost function which is based on the exact minimisation of the sum of weighted errors given by

$$J(k) = \sum_{l=1}^{k} \lambda^{k-l}(d(l) - \boldsymbol{w}^T(k)\boldsymbol{x}(l))^2 \tag{5}$$

where $\lambda \in (0, 1]$ is the forgetting factor which designates the memory of the algorithm. Equation (4) is therefore based on an underlying growing exponentially weighted window of the input data. The length of the window is given, to a first approximation, by $1/(1 - \lambda)$ [Haykin, 1996]. A unity value for the forgetting factor corresponds to infinite memory and is only suitable when statistical stationarity can be assumed. The update equation for the RLS algorithm is given by

$$\boldsymbol{w}(k+1) = \boldsymbol{w}(k) + \boldsymbol{R}^{-1}(k)\boldsymbol{x}(k)e(k) \tag{6}$$

where $\boldsymbol{R}{-}1(k)$ is the inverse of the deterministic correlation input matrix given by $\sum_{l=1}^{k} \lambda^{k-l}\boldsymbol{x}(l)\boldsymbol{x}^T(l)$. Notice the commonality between the adaptation algorithms, eqns. (1), (3) and (5), they are all based on the same form of update equation. The distinguishing feature is the form and complexity of the adaptation gain, for LMS it is a scalar constant, for NLMS it is a time varying scalar, whereas for RLS it is a time varying matrix.

Matrix inversion is an $O(L^3)$ operation, therefore it would not be feasible for a real-time algorithm to calculate a new value for $\boldsymbol{R}^{-1}(k)$ at each sample. However, this is not necessary. Some complexity reduction results from writing $\boldsymbol{R}(k) = \lambda\boldsymbol{R}(k-1) + \boldsymbol{x}(k)\boldsymbol{x}^T(k)$ which shows that the change in the data input matrix from one iteration to the next is a simple vector outer product of the most recent data vector, that is, a rank one matrix. The matrix inversion lemma [Haykin, 1996] can then be applied to obtain a simple update for $\boldsymbol{R}^{-}1(k)$ which does not require a matrix inversion, but just a division by a scalar, i.e.

$$\boldsymbol{R}^{-1}(k) = \frac{1}{\lambda}\Big[\boldsymbol{R}^{-1}(k-1) - \frac{\boldsymbol{R}^{-1}(k-1)\boldsymbol{x}(k)\boldsymbol{x}^T(k)\boldsymbol{R}^{-1}(k-1)}{\lambda + \boldsymbol{x}^T(k)\boldsymbol{R}^{-1}(k-1)\boldsymbol{x}(k)}\Big] \tag{7}$$

For convenience, the quantity $\boldsymbol{R}^{-1}(k)\boldsymbol{x}(k)$ in (5) is sometimes called the Kalman gain vector. Notice the RLS algorithm and Kalman filtering are very closely linked. The complete RLS algorithm is given by the next three equations

$$\boldsymbol{g}(k) = \frac{\boldsymbol{R}^{-1}(k-1)\boldsymbol{x}(k)}{\lambda + \boldsymbol{x}^T(k)\boldsymbol{R}^{-1}(k-1)\boldsymbol{x}(k)} \tag{8}$$

$$\boldsymbol{R}^{-1}(k) = \frac{1}{\lambda}[\boldsymbol{R}^{-1}(k-1) - \boldsymbol{g}(k)\boldsymbol{x}^T(k)\boldsymbol{R}^{-1}(k-1)] \tag{9}$$

$$\boldsymbol{w}(k+1) = \boldsymbol{w}(k) + \boldsymbol{g}(k)e(k) \tag{10}$$

To initialise the RLS algorithm, the most simple procedure is to use a soft constraint and set $\boldsymbol{R}^{-1}(0) = k\boldsymbol{I}$, that is, a scaled version of the identity matrix [Haykin, 1996]. When the forgetting factor is set to be less than unity, the effect of this soft constraint will quickly disappear. The coefficient vector is, as for the LMS algorithm, initialised as the zero vector.

The RLS algorithm has been written in Python below:

```python
def compute_error(x, d, w):
    return (d - np.dot(w, x))

def compute_kalman_gain(x, last_rinv, forget_factor):
    x = x[:, np.newaxis]
    a = np.matmul(last_rinv, x)

    return a/(forget_factor + np.matmul(x.T, a))

def compute_rinv(x, last_rinv, kalman_gain, forget_factor):
    x = x[:, np.newaxis]
    a = np.matmul(x.T, last_rinv)

    b = np.matmul(kalman_gain, a)
    return (1/forget_factor)*(last_rinv - b)

def rls(x, d, X, L, forget_factor):
    weights = [np.zeros(L)]
    z = np.zeros(L)
    rinv = np.eye(L)

    errors = []
    N = len(x)
    for k in range(0, N - L):
        x_k = x[k:k+L]
        d_k = x[k]
        w_k = weights[k]

        e_k = compute_error(x_k, d_k, w_k)

        kalman_gain = compute_kalman_gain(x_k, rinv, forget_factor)

        new_weights = w_k + kalman_gain.flatten()*e_k

        rinv = compute_rinv(x_k, rinv, kalman_gain, forget_factor)

        weights.append(new_weights)
        errors.append(e_k)


    return np.array(weights), np.array(errors)
```

3.1

Compare the code with equations (7), (8) and (9). Explain how the code implements the algorithm and how the algorithm is initialised. What is the complexity, in terms of number of additions and multiplications, of the RLS routines?

3.2

Repeat Exercises 1.1, 1.2 and 1.4 for the RLS algorithm with the forgetting factor set at unity.

3.3

Repeat the exercise above with the forgetting factor set at 0.99, 0.95 and 0.8.

3.4

Advanced challenge: How do the LMS, NLMS and RLS algorithms perform if you vary, with sample number, the coefficients of the unknown system? One example would be to fix the unknown system vector for 200 samples and then to change the coefficients at sample number 201 and keep these fixed for the next 200 samples (Read the help information for the command filter).

4 References

Haykin, S., "Adaptive Filter Theory", Third Edition, Prentice-Hall, 1996, Much improved on first and second editions, good sections on numerical issues in adaptive filtering and emerging adaptive techniques, particularly blind signal processing.

Widrow, B., and S.D. Steams, "Adaptive Signal Processing", Prentice Hall, 1985, was in the vanguard of adaptive filtering, introductory but strong on applications.

Bellanger, M., "Adaptive Digital Filters and Signal Analysis", Marcel Dekker, 1987. A firstclass book, he has really worked with adaptive filters.

Mandic, D. and Goh, V. "Complex Valued Nonlinear Adaptive Filters: Noncircularity, Widely Linear and Neural Models", Wiley, 2009.

5 Version history

- Version 1: Jonathon Chambers, July 1994.
- Version 2: Jonathon Chambers, November 1999.
- Version 3: Danilo Mandic and Eric Grinstein, October 2021