The Cooper Union Department of Electrical Engineering
Profs. Fred L. Fontaine
ECE416 Adaptive Algorithms
Problem Set I: Initial Experimentation
January 20, 2023

Here we are going to do some initial experiments, looking at an AR(2) process, and using both LMS and RLS to estimate the AR parameters.
We are looking at an AR(2) process whose innovations filter has a pair of real poles. You are going to look at 3 cases (i.e., repeat everything for each case):

$$p = (0.7, 0.6)\,,\ (0.9, 0.8)\,,\ (0.9, -0.8)$$

The innovations filter is:

$$H(z) = \frac{1}{(1 - p_1 z^{-1})(1 - p_2 z^{-1})} = \frac{1}{1 + a_1 z^{-1} + a_2 z^{-2}}$$

where p_1, p_2 are the poles. This will correspond to the model:

$$x[n] = v[n] - a_1 x[n-1] - a_2 x[n-2]$$

Here we will assume the innovations $v[n]$ has unit variance $\sigma_v^2 = 1$. The notes on rational power spectral densities provides an explicit formula for the correlation function.
We want to set up an adaptive algorithm to determine the model parameters. We do this by setting up an M^{th} order linear prediction problem, with $M \times 1$ tap weight vector $\mathbf{w}$, whose "optimal" value (for $M \geq 2$) is:

$$\mathbf{w}_{\text{opt}} = \begin{bmatrix} -a_1 \\ -a_2 \\ \mathbf{0}_{(M-2)\times 1} \end{bmatrix}$$

For the adaptive filters, the desired signal $d[n]$ and input vector $\mathbf{u}[n]$ are given by:

$$\begin{aligned} d[n] &= x[n] \\ \mathbf{u}[n] &= \mathbf{x}_M[n-1] = \begin{bmatrix} x[n-1] \\ x[n-2] \\ \vdots \\ x[n-M] \end{bmatrix} \end{aligned}$$

- I prescribed three pole pairs above. Repeat EVERYTHING for each pole set.
- When you run the adaptive algorithms, we are going to take $M = 2, 4, 10$. We know the "right answer" is second order, but in general we don't know that in advance. You will look at what happens when the model order we choose for the adaptive filter is too large.
- When you run the algorithms, start with a small number of iterations just to make sure you aren't making errors, or the system is too slow. Maybe start with $N_{iter} = 50$. You can eventually boost the number of iterations N_{iter} up to 1000 or even 2000 if things are "still happening".

Define R_L as the $L \times L$ correlation matrix of $\mathbf{x}_L[k]$, with $r[m] = E(x[n+m]x^*[n])$ the autocorrelation function of x.

$$R_L = E\left(\mathbf{x}_L[k]\mathbf{x}_L^H[k]\right)$$

The ij^{th} element is:

$$\begin{aligned}(R_L)_{ij} &= E(x[k-i+1]x^*[k-j+1]) \\ &= r[(k-i+1)-(k-j+1)] \\ &= r[j-i]\end{aligned}$$

Also, since $r[m] = r^*[-m]$, we get R_L is Hermition with constants on the diagonal, what is called *Toeplitz symmetry.* For example:

$$R_4 = \begin{bmatrix} r[0] & r[1] & r[2] & r[3] \\ r[-1] & r[0] & r[1] & r[2] \\ r[-2] & r[-1] & r[0] & r[1] \\ r[-3] & r[-2] & r[-1] & r[0] \end{bmatrix}$$

Actually, since here all the data is real, we have $r[-m] = r[m]$.

Data Analysis

Before running the adaptive filters, we will perform some analysis on the data. I prescribed three possible pole sets. Repeat EVERYTHING for each case.

Task 1: Synthesize The Random Signal

We want to generate N_0 samples of x for us to use. If you generate N samples of v, then *x=filter(1,a,v);* will generate N samples of x. However, if we are going to assume that x is stationary, there is a problem during the initialization period. If we start at $n = 1$, in effect this process treats $x[n] = 0$ for $n \leq 0$, and the result is this generated x will only be asymptotically stationary. We expect a time constant determined by the dominant pole, i.e., a transient behavior $\sim |p|_{\max}^n$ where $|p|_{\max}$ is the maximum of the pole magnitudes. One way to generate data that is closer to stationary according to our idealized model is to create extra points to be discarded later, say N_{init} points. That is, make v length $N_0 + N_{\text{init}}$, and after generating x, discard the first N_{init} values. If we want the transient to decay to say 0.01, i.e., $|p|_{\max}^{N_{init}} \approx 0.01$, you can solve for N_{init}. This determines N_{init}.

After removing the first N_{init} samples, we should work with the renumbered data $x[1], \cdots, x[N_0]$. If we prescribe the numer of iterations N_{iter} to run our algorithm, that will determine N_0. (Note $N_0 \neq N_{\text{iter}}$ in general). What I mean is, if we stack the data fed into the adaptive filter across iterations as columns in a matrix, it has the form:

$$X = \begin{bmatrix} x[M+1] & x[M+2] & \cdots & x[N_0] \\ x[M] & x[M+1] & \cdots & x[N_0-1] \\ \vdots & \vdots & & \\ x[1] & x[2] & \cdots & x[N_0-M] \end{bmatrix} \tag{1}$$

The top row is the desired signal across time, the bottom M rows is the input vector $\mathbf{u}$ across time. Note in this form we only use data that is actually available to us. There are

alternatives that can be used as times, such as *prewindowing:*

$$X = \begin{bmatrix} x[1] & \cdots & x[M] & x[M+1] & \cdots & x[N_0] \\ 0 & \ddots & x[M-1] & x[M] & \cdots & x[N_0-1] \\ \vdots & & \vdots & \vdots & & \\ 0 & \cdots & x[1] & x[2] & \cdots & x[N_0-M+1] \\ 0 & \cdots & 0 & x[1] & \cdots & x[N_0-M] \end{bmatrix}$$

or for prewindowing we can use the data we discarded. Similarly, sometimes we can use postwindowing, extending x on the right say with zeros. In any case, here we will stay with the form in (1). Whatever the form of X, the number of iterations N_{iter} should match the number of columns. From (1), you should see how to determine N_0 from M and N_{iter}.
Given: Poles p, order M, the factor 0.01 indicated above, and N_{iter}.
Generate: X as in (1). [What I mean is create code to do this, given the parameters above. You will run it for specific values indicated later] **Hint:** Once you have the data set $x[n]$ for $1 \le n \le N_0$ (i.e., after you clip off the initialization portion), use MATLAB *toeplitz* to generate the matrix X in **one line!**
Task 2: Analyze the Random Signal
We first want to look at $r[\ell]$ for lags $0 \le \ell \le L$, and the PSD $S(\omega)$.
Given: Poles p, maximum lag L.
Generate: Using the theoretical formula for the correlation, compute $r[\ell]$ for $0 \le \ell \le L$. Then arrange this in an $(L+1)\times(L+1)$ Toeplitz matrix R_{L+1}. Also, compute and plot the PSD for $-\pi \le \omega \le \pi$. **Remark:** If you use the formula given in the rational PSD notes, because of numerical error you may end up with small imaginary parts in S; check that any imaginary part is small, first, and then use *S=real(S)* to get rid of that.
In what follows, we will use $L = 20$. Also, I prescribed three possible pole sets, run through everything in each case.
We want to check certain things:

1. Compute the eigenvalues of $R_{L\times 1}$, verify they are positive real, and stem plot them in descending order.

2. Let R_m denote the $m \times m$ submatrix of R_{L+1} (upper left corner, i.e., first m rows and columns); as a special case, $R_1 = r[0]$, just the scalar. For example, $R_2 = \begin{bmatrix} r[0] & r[1] \\ r[1] & r[0] \end{bmatrix}$. For the chain of submatrices, $1 \le \ell \le L+1$, compute the sequence of $\lambda_{\min}$ and $\lambda_{\max}$, and in particular you should notice a pattern here! **Remark:** As $L \to \infty$, $\lambda_{\min} \to S_{\min}$ and $\lambda_{\max} \to S_{\max}$.

3. Compute and plot $S(\omega)$. Also determine $S_{\min}$, $S_{\max}$ and verify that in all cases $S_{\min} \le \lambda_{\min}$ and $\lambda_{\max} \le S_{\max}$.

4. We now want to compute an estimate of r from the data. Rather than calling MATLAB functions for it, we are going to do it directly from the data. Again, given $x[n]$ for $1 \le n \le N_0$, form (1) with $M = L$. Then:

$$R_{L+1} \approx \frac{1}{K} X X^T$$

What value of K makes this unbiased? Use that value! Here, we want the number of columns of X to be 1000 for purposes of computing this estimate. Look at your estimated matrix $\hat{R} = \frac{1}{K} X X^T$. Is it exactly Toeplitz? Use the top row as your estimate of the correlation, $\hat{r}[\ell]$ for $0 \leq \ell \leq L$. Superimpose stem plots of $r[\ell]$ and $\hat{r}[\ell]$ for $0 \leq \ell \leq L$.

5. Suppose we compute the singular values of X; they provide the same information as the eigenvalues of $\hat{R} = \frac{1}{K} X X^T$. How are they related? [You have to account for the scaling factor $1/K$ here!] Viewed another way, if we want to perform SVD analysis, it should be done on a scaled matrix αX. What is the appropriate choice for α?

6. As another check, we expect $r[0] = \frac{1}{2\pi} \int_{-\pi}^{\pi} S(\omega)\, d\omega \approx \frac{1}{2\pi} \sum S_i \Delta\omega_i$ where $\Delta\omega_i$ is the frequency spacing corresponding to the vector of computed PSD values S_i you have. Compute this numerical approximation to the integral and compare to the "theoretical" $r[0]$.

LMS Algorithm

As noted above, for each pole set, we want to run LMS for $M = 2, 4, 10$.
Let $\mathbf{w}_{\text{opt}}$ denote the optimal tap weight vector, and $\mathbf{w}[n]$ the value computed at iteration n, and $e[n]$ the error at iteration n.
The mean-square error is:

$$J[n] = E\left(|e[n]|^2\right)$$

In practice, this is estimated through a process described here, and the graph of the resulting $J(n)$ is called the *learning curve.*
We are also interested in:

$$D[n] = E\left(\|\mathbf{w}[n] - \mathbf{w}_{\text{opt}}\|^2\right)$$

which is estimated in a similar scheme, resulting in $D(n)$ called the *mean-square deviation.*
Recall that in the problem we are setting up, the theoretical minimum value of $J[n]$ is denoted $J_{\min}$, and is given by:

$$J_{\min} = \sigma_v^2$$

where σ_v^2 is the variance of the innovations $v[n]$.
What will happen, when the algorithm "works", is the learning curve will (kind of) converge to a steady state value we will denote $J(\infty)$. The curve itself will be "noisy" however; what happens is the mean value (the center of the curve) with converge to limit $J(\infty) > J_{\min}$, but the variance of the learning curve does not go to 0. This is called *convergence in the mean.* The *misadjustment* is defined as:

$$\mathcal{M} = \frac{J(\infty)}{J_{\min}}$$

The value $J(\infty)$ can be estimated by averaging points from the learning curve $J(n)$ for large n (after convergence seems to have occurred).
The reason for misadjustment is that, even if the adaptive filter at some point reaches the "ideal" $\mathbf{w}_{\text{opt}}$, there will still be a nonzero error, and algorithm will keep changing $\mathbf{w}[n]$. At

any given moment, even after the algorithm has "converged", the best we can hope for is that $\mathbf{w}[n]$ is close to $\mathbf{w}_{\text{opt}}$, but it will still continue to vary and mismatch the ideal. That will cause the *excess error* $J(\infty) - J_{\min}$. In any case, we can see the misadjustment $\mathcal{M} \geq 1$. An unstable algorithm can be viewed as the case $\mathcal{M} = \infty$.

When the algorithm works, the learning curve will decay at an exponential rate called the *rate of convergence.* Don't get too excited– there may be a VERY slow time constant. In fact, when you run LMS I suggest you use $N_{\text{iter}} = 1000$.

So how do you formulate the learning curve and mean-square deviation curve? Repeat the experiment (generate independent data each time) with say $K_0 = 100$ runs. Let $e_k[n]$ denote the error at *iteration* index n, for the k^{th} run of the experiment. Then:

$$J[n] = \frac{1}{K_0} \sum_{k=1}^{100} |e_k[n]|^2$$

Similarly, if $\mathbf{w}_k[n]$ is the tap-weight vector computed at iteration n during the k^{th} run of the experiment, then:

$$D[n] = \frac{1}{K_0} \sum_{k=1}^{100} \|\mathbf{w}_k[n] - \mathbf{w}_{\text{opt}}\|^2$$

Increasing μ, the step-size, with improve the rate of convergence, but if it gets too big it will cause *instability.* The theoretical condition for stability is:

$$0 < \mu < \frac{2}{\lambda_{\max}}$$

where $\lambda_{\max}$ is the maximum eigenvalue of $E\left(\mathbf{u}[n]\,\mathbf{u}^H[n]\right)$. It is common to take the midpoint, say $\mu \approx \frac{1}{\lambda_{\max}}$. Sometimes it is easier to use as a starting point $\mu = \frac{1}{S_{\max}}$. **COMMENT: VERIFY THAT** $\mu = \frac{1}{S_{\max}}$ SATISFIES THE STABILITY CONDITION! This stability condition is not exact, and is based upon an imprecise analysis.

Task: For each of the pole sets, for each of the values $M = 2, 4, 10$, repeat the following.

When you start testing LMS, use $1/S_{\max}$ as a reference point. Try say values $0.1\frac{1}{S_{\max}}$ and $\frac{1}{S_{\max}}$, and run the algorithm as stated above. If your learning curve seems to blow up or have ridiculously large spikes, that means the algorithm is unstable. Play around with different μ until you find a kind of boundary of stability.

You should observe a faster rate of convergence for larger μ though. Try a value of μ that is "safe", say 0.5 times the stability boundary you seem to sense, and another value say 0.05 times that stability boundary. Superimpose graphs of the learning curve, and superimpose graphs of the mean-square deviation. In the learning curve, also draw a horizontal line at $J_{\min}$ to use that as a nice reference.

As suggested, you could aim for $N_{\text{iter}} = 1000$, but feel free to go higher or lower based on how fast the algorithm seems to converge. [Use one N_{iter} for every μ you end up using, however, for comparison purposes].

Compute the misadjustment by estimating $J(\infty)$ by averaging $J[n]$ over say the last 50 or so iterations.

Also, to see why we are doing 100 runs, separately do one graph of $|e[n]|^2$ and one graph of $\|\mathbf{w}[n] - \mathbf{w}_{\text{opt}}\|^2$ corresponding to just ONE RUN of the algorithm.

Comments: In many cases, the misadjustment goes UP as the rate of convergence improves. The reason is higher μ leads to larger changes at each step, so convergence happens faster (assume stability is kept). However, at steady-state, the larger μ also means $\mathbf{w}[n]$ jumps around more, increasing misadjustment. You may not observe that result here, as this is such a clean problem, on the one hand, and the theory is really not universally applicable (only an approximate analysis based on a number of assumptions is possible). In any case, make COMMENTS ON:

- How μ affects rate of convergence and misadjustment.
- How M affects the above, and also the range of μ that need to be used (e.g., does the higher M seem to change the boundary of stability for μ).
- How the poles affect the above. [When you change the poles you are changing $S_{\max}$. So what I mean here is to what extent is $1/S_{\max}$ a reasonable value to use in the algorithm]

Finally, we want to look at $\mathbf{w}$. Ideally, the first two elements are $-a_1, -a_2$, and the rest are 0. Is this true for any particular run? Try this: take say the final $\mathbf{w}_k[N_{\text{iter}}]$ found for each of the K_0 runs, and average them (i.e., average the 1st coefficient around $K_0 = 100$ runs, is it close to a_1?

Remark: I am implying here the condition for stability has the form $0 < \mu < \mu_{\max}$. Under more complicated situations, such as consideration of finite precision effects, it is not always that simple. However, at least for this experiment with a simple model with a high-precision computational platform (IEEE 64-bit floating point), this simple stability condition should apply.

Remark: It seems choosing μ is best done when we already know the answer, unfortunately. The solution is to pick a μ small enough that we are comfortable the algorithm remains stable for "most" situations we expect to encounter, and if we try higher μ the algorithm may work better and faster but we also run a higher risk of instability. Of course, the best solution is to use a more sophisticated algorithm that, for example, adaptively updates μ in some way. Indeed, most adaptive algorithms are variants of LMS or RLS, with LMS certainly the "quickest" way to get something up and running before migrating to a more refined approach.

RLS Algorithm

For the RLS algorithm, we are going to take $\lambda = 0.9$. At least as given, the only other parameter to consider is the value of δ used in the initial condition (i.e., $P[0] = \delta^{-1} I$). Here, use $\delta = 1$.

One thing to note is in the RLS algorithm, the iteration for $P(n)$ has the form:

$$P(n) = P(n-1) - [\text{stuff}]$$

where *stuff* is a Hermitian matrix (by construction). Stability of the algorithm very much hinges on $P(n)$ remaining *positive definite.* To be more precise, the update expresses $P(n)$ as the *difference* between a positive definite matrix and a non-negative definite matrix. This will not necessarily be positive definite in general, but HERE the result is that indeed the

$P(n)$ has a guarantee to be pd. Until limited numerical precision is accounted for! This is a critical problem in RLS and, more broadly, the Kalman filter (of which RLS is a special case). [Later we will see techniques to overcome this problem]
Insert code in your algorithm so that at each iteration it checks of P is indeed positive definite, and have it report out if this condition ever fails as you run the experiment below.]

As before, we want to generate the learning curve and mean-square deviation curve. Do this for each choice of p vector, and each choice of M. Start with $N_{\text{iter}} = 50$, and increase it as needed until you observe convergence. You probably won't need to go much beyond $N_{\text{iter}} = 100$.
Compute the misadjustment (well, estimate it since, as noted above, the learning curve retains a nonzero variance around a limiting value $J(\infty)$).
The rate of convergence should be much faster than LMS, and the algorithm is stable, and there is no mysterious μ to tweak. How is the misadjustment compared to LMS?
Look at the steady-state tap weight vector as well (as you did for LMS).

When the RLS algorithm works well (as it should here), it can vastly outperform LMS. Although it seems that LMS has an additional problem of estimating a suitable hyperparameter μ, LMS does have significant advantages primarily in terms of its robustness when our models are not as precise and "clean."