Adaptive Array Signal Processing

[5SSC0]

Assignment Part 1A: Adaptive Algorithms

REPORT

Group number: 2

Names including ID:
1: Hengjian ZHANG (1654381)
2: Fer RADSTAKE (0679273)

Date: 12/02/2021

1.2 Scenario 1: Known statistics

1.2.1 Wiener filter

a

$E(r^2[k]) = E\left\{(e[k] - \underline{w}^t \cdot \underline{x}[k]) \cdot (e[k] - \underline{x}^t[k] \cdot \underline{w})\right\}$
$= E\left\{e^2[k]\right\} - \underline{w}^t E\{\underline{x}[k]e[k]\} - E\left\{e[k]\underline{x}^t[k]\right\}\underline{w} + \underline{w}^t E\left\{\underline{x}[k]\underline{x}^t[k]\right\}\underline{w}$
$= E\left\{e^2[k]\right\} - \underline{w}^t \underline{r}_{ex} - \underline{r}_{ex}^t \underline{w} + \underline{w}^t R_x \underline{w}$
Let $\underline{\nabla} = \frac{\mathrm{d}J}{\mathrm{d}\underline{\mathrm{w}}} = -2\left(\underline{r}_{ex} - \mathrm{R}_x\underline{\mathrm{w}}\right) = \underline{0}$,
$\underline{\mathbf{w}}_0 = \arg_{\underline{\mathbf{w}}} \min E\left(r^2[k]\right) = \mathrm{R}_x^{-1} \cdot \underline{r}_{ex}$
We know $\mathbf{R}_x = \begin{pmatrix} 2 & -1 \\ -1 & 2 \end{pmatrix}; \underline{\mathbf{r}}_{ex} = \begin{pmatrix} 0 \\ 3 \end{pmatrix}$ So $\underline{\mathbf{w}}_0 = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$

b

$\underline{r}_{xr} = E\{\underline{x}[k] \cdot r[k]\}$
$= E\{\underline{x}[k] \cdot (e[k] - \underline{x}^t[k] \cdot \underline{w})\}$
$= E\{\underline{x}[k]e[k]\} - E\left\{\underline{x}[k]\underline{x}^t[k]\right\}\underline{w}$
$= \underline{r}_{ex} - \mathrm{R}_x\underline{\mathrm{w}}$
when $\underline{\mathrm{w}} = \underline{\mathbf{w}}_0$, $\underline{r}_{xr} = \underline{0}$.
This means $x[k]$ and $r[k]$ are entirely uncorrelated.

c

Use time-averaging (ergodicity):

$$\hat{\mathrm{R}}_x = \frac{1}{L}\sum_{i=0}^{L-1} \underline{x}[k-i] \cdot \underline{x}^t[k-i] = \frac{1}{L}\mathrm{X}^t \cdot \mathrm{X} = \frac{1}{L}\overline{\mathrm{R}}_x$$

$$\hat{\underline{r}}_{ex} = \frac{1}{L}\sum_{i=0}^{L-1} \underline{x}[k-i] \cdot e[k-i] = \frac{1}{L}\mathrm{X}^t \cdot \underline{\mathrm{e}} = \frac{1}{L}\overline{\mathrm{r}}_{ex}$$

Here, we only use the sample data to estimate the $\hat{\mathrm{R}}_x$ and $\hat{\underline{r}}_{ex}$, but only when $L \to \infty$, $\underline{\mathrm{w}}_{\mathrm{ls}} = \underline{\mathrm{w}}_{mmse}$. So basically we only used limited samples to represent the whole feature in order to get $\hat{\mathrm{R}}_x$ and $\hat{\underline{r}}_{ex}$.

1.2.2 Steepest Gradient Descent

d

For SGD algorithm:

$$\underline{\mathrm{w}}[k+1] = \underline{\mathrm{w}}[k] + 2\alpha\left(\underline{r}_{ex} - \mathrm{R}_x\underline{\mathrm{w}}[k]\right)$$

when the SGD reaches steady-state, we have:

$$\begin{aligned}&\underline{\mathrm{w}}[k+1] \simeq \underline{\mathrm{w}}[k] \simeq \underline{\mathrm{w}}[\infty]\\ \Rightarrow&\underline{\mathrm{w}}[\infty] \simeq \underline{\mathrm{w}}[\infty] + 2\alpha\left(\underline{r}_{ex} - \mathrm{R}_x\underline{\mathrm{w}}[\infty]\right)\\ \Rightarrow&\underline{\mathrm{w}}[\infty] \simeq \mathrm{R}_x^{-1}\cdot\underline{r}_{ex}\end{aligned}$$

Thus,

$$\lim_{k\to\infty}\underline{w}[k] \simeq R_x^{-1}\cdot\underline{r}_{ex} = \begin{pmatrix}1\\2\end{pmatrix}$$

e

Define difference weight vector: $\underline{\mathrm{d}}[k] = \underline{\mathrm{w}}[k] - \underline{\mathrm{w}}_o$

$$\begin{aligned}\underline{\mathrm{w}}[k+1] &= \underline{\mathrm{w}}[k] + 2\alpha\left(\underline{\mathrm{r}}_{\mathrm{ex}} - \mathrm{R}_{\times}\underline{\mathrm{w}}[k]\right)\\ \underline{\mathrm{w}}[k+1] - \underline{\mathrm{w}}_o &= (\mathrm{I} - 2\alpha\mathrm{R}_x)\cdot\underline{\mathrm{w}}[k] - \underline{\mathrm{w}}_o + 2\alpha\underline{\mathrm{r}}_{ex}\\ \Rightarrow \underline{\mathrm{d}}[k+1] &= (\mathrm{I} - 2\alpha\mathrm{R}_x)\cdot\underline{\mathrm{d}}[k]\end{aligned}$$

Recursion:

$$\underline{\mathrm{d}}[k] = (I - 2\alpha\mathrm{R}_x)\cdot\underline{\mathrm{d}}[k-1] = \cdots = (I - 2\alpha\mathrm{R}_x)^k\cdot\underline{\mathrm{d}}[0]$$

Stable iff:

$$\lim_{k\to\infty}(I - 2\alpha\mathrm{R}_x)^k = 0$$

Use eigenvalue decomposition :
With $\mathrm{Q}^h\cdot\mathrm{Q} = \mathrm{Q}\cdot\mathrm{Q}^h = 1$ and $\mathrm{R}_x = \mathrm{Q}\wedge\mathrm{Q}^h$

$$\begin{aligned}\Rightarrow (I - 2\alpha\mathrm{R}_x)^k &= \left(\mathrm{Q}\mathrm{Q}^h - 2\alpha\mathrm{Q}\wedge\mathrm{Q}^h\right)^k\\ &= \mathrm{Q}(I - 2\alpha\wedge)^k\mathrm{Q}^h\end{aligned}$$

Change of variables: $\underline{\mathrm{D}}[k] = \mathrm{Q}^h\cdot\underline{\mathrm{d}}[k]$

$$\underline{\mathrm{d}}[k] = (I - 2\alpha\mathrm{R}_x)^k\,\underline{\mathrm{d}}[0] \Rightarrow \underline{\mathrm{D}}[k] = (I - 2\alpha\Lambda)^k\underline{\mathrm{D}}[0]$$

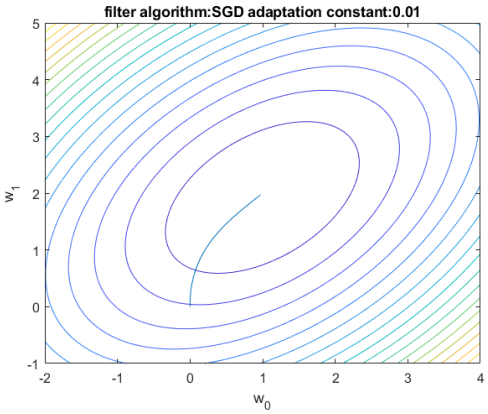
Recursion stable iff: $\lim_{k\to\infty}(I - 2\alpha\Lambda)^k = 0$
Both matrices I and $\wedge$ diagonal
$\Rightarrow$ Stable iff: $|I - 2\alpha\lambda_i| < 1 \quad\Leftrightarrow\quad 0 < \alpha < \frac{1}{\lambda_i}$ for $i = 0, 1, \cdots, N-1$
Thus SGD algorithm stable iff: $0 < \alpha < \frac{1}{\lambda_{\max}}$.

It can be calculated that the eigenvalues of R_x are 1 and 3; therefore SGD is stable iff $0 < \alpha < \frac{1}{3}$.

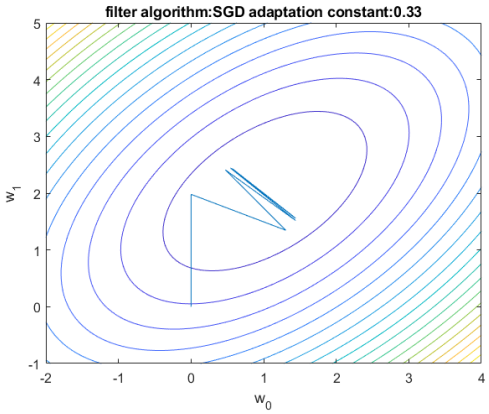
f

SGD filter update MATLAB code:

```
Rx = [2 -1; -1 2]; % see 1.2.1.a
rex = [0;3]; % see 1.2.1.a
filter.w = w_old + 2 * alpha * (rex - Rx*w_old);
```



(a) $\alpha = 0.01$



(b) $\alpha = 0.33$

Figure 1: Convergence of w_1 and w_2 for two values of α.

The resulting convergence of w_1 and w_2 are shown in Fig.1 for values $\alpha = 0.1 \ll \frac{1}{3}$ and $\alpha = 0.33 < \frac{1}{3}$. It is clear that with $\alpha = 0.33$ (only slightly less than the maximum stable α), w_1 and w_2 converge relatively fast but tend to overshoot the optimum value. It can also be seen that this overshoot mainly happens in the direction of steepest descent. It is clear that this overshoot doesn't happen for $\alpha = 0.01$ as seen in Fig. 1a. Also worthy of note is the difference in convergence speed for w_0 and w_1 where w_1 converges faster. This is also apparent from the gradient, where the lines are closer together in the direction of w_1 than in the direction of w_0.

1.2.3 Newton's Method

g

Newton's Method: $\underline{w}[k+1] = \underline{w}[k] - \alpha R_x^{-1} \underline{\nabla} \Rightarrow$

$$\underline{w}[k+1] = \underline{w}[k] + 2\alpha R_x^{-1} \cdot (\underline{r}_{ex} - R_x \underline{w}[k])$$

Convergence Newton:

$$\underline{\mathrm{d}}[k+1] = \left(\mathrm{I} - 2\alpha \mathrm{R}_x^{-1} \mathrm{R}_x\right) \underline{\mathrm{d}}[k] = (1 - 2\alpha) \underline{\mathrm{d}}[k]$$

Here, all filter weights converge at the same rate of $1 - 2\alpha$

h

To make Newton's method stable, from g, we have $0 < (1 - 2a) < 1$,

$$\Rightarrow \text{ Convergence } 0 < \alpha < 0.5$$

i

Newton filter update MATLAB code:

```
Rx = [2 -1; -1 2]; % see 1.2.1.a
rex = [0;3]; % see 1.2.1.a
filter.w=filter.w + 2 * alpha * (Rx^-1) * (rex - Rx*filter.w);
```

This filter was run on the generated signal and the result shown in Fig. 2. It is apparent that the convergence is now a straight line, as opposed to Fig. 1a where the convergence is a curve. From this, it can be concluded that indeed all filter weights converge at the same rate, as was shown in 1.2.3.g.

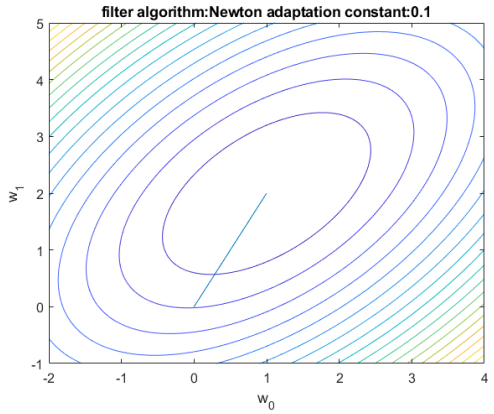


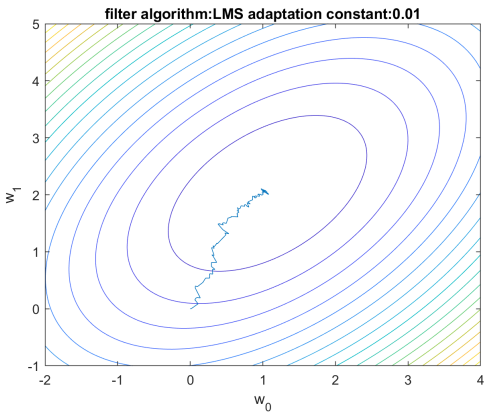
Figure 2: Convergence of the Newton algorithm

1.3 Scenario 2: Unknown statistics

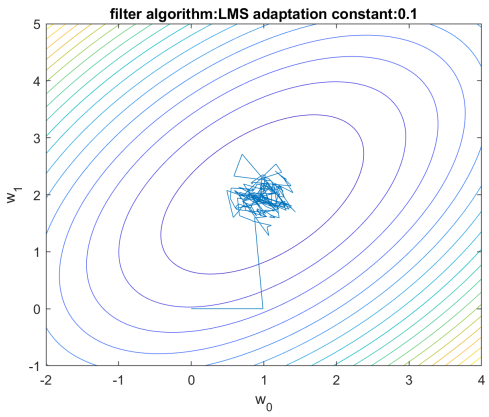
1.3.1 LMS and NLMS

a

```
filter.w=filter.w+2*alpha*x*r;
```



(a) $\alpha = 0.01$



(b) $\alpha = 0.1$

Figure 3: Convergence of w_1 and w_2 for two values of α.

Tradeoff choosing α: It is clear in Fig.3 that the bigger the α, the faster the w converge towards the optimal, while this also tend to overshoot. When $\alpha = 0.01$, the step size is smaller but it also doesn't really overshoot. So we need to choose between convergence speed and overshoot.

b

```
In adaptive_filter.m:
properties
sigma_x;
function obj = adaptive_filter(length,type,adaptation_constant)
obj.sigma_x = 0;
function obj = filter(obj,x,e)
obj.sigma_x = mean(obj.x_delayed.^2);
In update_filter.m:
sigma_x=filter.sigma_x;
filter.w=filter.w+2*alpha/sigma_x*x*r;
```

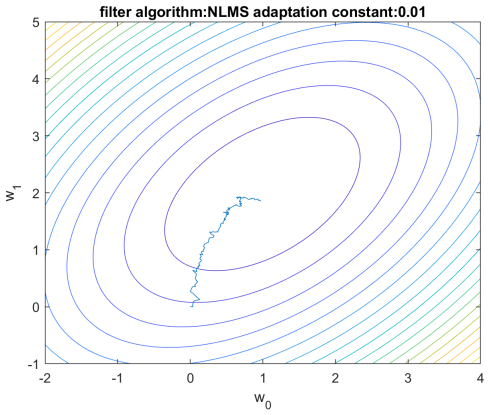


Figure 4: NLMS for $\alpha = 0.01$

By applying normalization factor, the convergence rate is normalized. The maximum convergence rate of the NLMS algorithm will always be faster than the LMS algorithm if the eigenvalues are unequal.

1.3.2 RLS and FDAF

c

LS and Newton both have the disadvantage that they require R_x^{-1}.
SGD requires known R_x and $\underline{r}_{ex}$.
LMS and NLMS has the problem that convergence gradient based algorithms relies on coloration input: $\underline{\nabla} = -2\left(\underline{r}_{ex} - \mathrm{R}_x\underline{\mathrm{w}}[k]\right)$.

RLS solves this problem by taking into account just the current and the previous sample and a scaled version of the previous estimate of R_x and r_{ex} (this scaling factor automatically leads to the exponential sliding window as seen in the course slides). This makes RLS especially suitable for real-time applications.

FDAF can be seen as a frequency domain version of LMS. As whitening in the frequency domain is cheap, FDAF has less computational complexity (especially for large filter lengths).

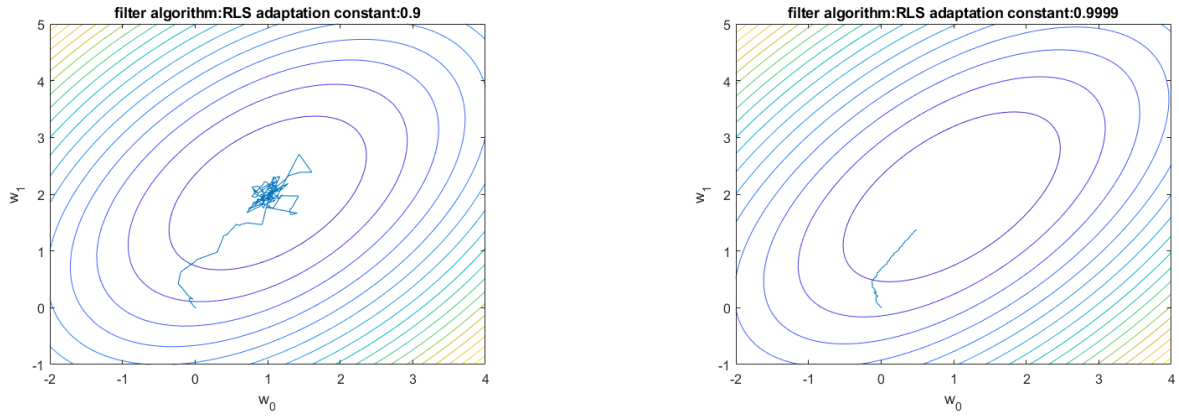
d

RLS filter update MATLAB code:

```
1    gamma=filter.adaptation_constant;
2    g=(filter.Rx_inv * filter.x)/(gamma^2 + filter.x'*filter.
        Rx_inv*filter.x);
```

```
    filter.Rx_inv = gamma^-2 * (filter.Rx_inv - g*filter.x'*
       filter.Rx_inv);
    filter.rex = gamma^2 * filter.rex + filter.x * filter.e
    filter.w = filter.Rx_inv * filter.rex;
```

This filter was run on the generated signal and the result shown in Fig. 5 (both with the same initial estimation for R_x). It can be seen that if γ increases (i.e. more "memory" taken into account), the convergence is smoother and improves over time, whereas if γ decreases, the convergence is faster.



(a) $\gamma = 1 - 10^{-1}$ (b) $\gamma = 1 - 10^{-4}$

Figure 5: Convergence of w_1 and w_2 for two values of γ.

e

FDAF filter update MATLAB code:

```
    alpha=0.1;
    beta = 0.5;
    X = filter.F * filter.x_delayed;
    Xi = X.*eye(filter.length);
    filter.P = beta * filter.P + (1-beta) * abs(Xi)^2 / filter.
       length;
    filter.W = filter.W+ 2 * alpha*filter.P^-1 * conj(X) * r;
    filter.w = filter.F*filter.W;
```

This filter was run on the generated signal and the result shown in Fig. 6 for $\alpha = 0.1$, $\beta = 0.5$.

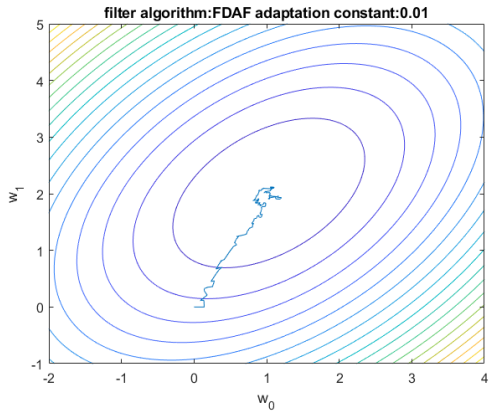


Figure 6: Convergence of the FDAF algorithm

f

	Computational complexity	Convergence speed/stability/accuracy
LMS	2	3
NLMS	3	2
RLS	4	1
FDAF	1	4

NLMS adds normalization to the LMS algorithm, which trades a bit of extra computational complexity for increased accuracy.

The advantage of RLS is very fast convergence. However its computational complexity is the largest of the four algorithms being $O(N^2)$ whereas the others are $O(2N)$.

FDAF decorrelates the different filter coefficients by flattening in the frequency domain, which is a very efficient procedure, especially for large filter lengths. This makes it rather fast, but the extra operations introduce small errors which hinder its convergence.

Appendix

adaptive_filter.m

```
classdef adaptive_filter
    %Performs sample-wise adaptive filtering
    %   input x and e, outputs r

    properties
        length;  %number of filter coefficients
        type;    %filter type
        adaptation_constant;
        w=[];           %filter coefficients
        x_delayed = []; %delayed input signals
        w_history = []; %keep track of filter coefficients as
           they evolve
        r;

        %add your own variables here
        sigma_x;
        F;
        F_inv;
        P;
        W;

    end

    methods
        function obj = adaptive_filter(length,type,
           adaptation_constant)
            if(nargin>0)
            obj.length=length;
            obj.type=type;
            obj.adaptation_constant=adaptation_constant;
            obj.w=zeros(length,1);
            obj.x_delayed=zeros(length,1);
            obj.w_history=zeros(0,length);

            %initialize your variables here
            obj.sigma_x = 0;
            obj.F=dftmtx(length);
            obj.F_inv = obj.F^-1;
            obj.P = eye(length);
            obj.W = obj.F_inv * obj.w;
```

```
39
40              end
41          end
42          function obj = filter(obj,x,e)
43              obj.x_delayed=[x ; obj.x_delayed(1:obj.length-1)];
44              obj.sigma_x = mean(obj.x_delayed.^2);
45              e_hat=obj.w.' * obj.x_delayed;
46              obj.r=e-e_hat;
47              obj.w_history=[obj.w_history ; obj.w.']; %you may
                    want to remove this line to gain speed
48              obj=update_filter(obj,e);
49          end
50      end
51
52  end
```

update_filter.m

```
1   function [ filter ] = update_filter( filter,e )
2
3   x=filter.x_delayed;
4   w_old = filter.w;
5   r=filter.r;
6   filter_type=filter.type;
7   sigma_x=filter.sigma_x;
8   filter.w=w_old; %default output
9
10  %% A1 scenario 1:f
11  if strcmpi(filter_type,'SGD')
12      %implement the SGD update rule here
13      alpha=filter.adaptation_constant;
14      Rx = [2 -1; -1 2]; % see 1.2.1.a
15      rex = [0;3]; % see 1.2.1.a
16      filter.w = filter.w + 2 * alpha * (rex - Rx*w_old);
17  end
18
19  %% A1 scenario 1:i
20  if strcmpi(filter_type,'Newton')
21      %implement the Newton update rule here
22      alpha=filter.adaptation_constant;
23      Rx = [2 -1; -1 2]; % see 1.2.1.a
24      rex = [0;3]; % see 1.2.1.a
25      filter.w=filter.w + 2 * alpha * (Rx^-1) * (rex - Rx*w_old);
```

```
end

%% A1 scenario 2:a
if strcmpi(filter_type,'LMS')
    %implement the LMS update rule here
    alpha=filter.adaptation_constant;
    filter.w=filter.w+2*alpha*x*r;
end

%% A1 scenario 2:b
if strcmpi(filter_type,'NLMS')
    %implement the NLMS update rule here
    alpha=filter.adaptation_constant;
    filter.w=filter.w+2*alpha/sigma_x*x*r;
end

%% A1 scenario 2:d
if strcmpi(filter_type,'RLS')
    %implement the RLS update rule here
    lambda=filter.adaptation_constant;
    gamma=1-1e-4;
    Rx_inv_old = filter.Rx_inv;
    g=(Rx_inv_old * filter.x)/(gamma^2 + filter.x'*Rx_inv_old*
       filter.x);
    filter.Rx_inv = gamma^-2 * (Rx_inv_old - g*filter.x'*
       Rx_inv_old);
    filter.rex = gamma^2 * filter.rex + filter.x * filter.e
    filter.w = filter.Rx_inv * filter.rex;
end

%% A1 scenario 2:e
if strcmpi(filter_type,'FDAF')
    %implement the FDAF update rule here
    alpha=filter.adaptation_constant;
    beta = 0.5;
    X = filter.F * filter.x_delayed;
    Xi = X.*eye(filter.length);
    filter.P = beta * filter.P + (1-beta) * abs(Xi)^2 / filter.
       length;
    filter.W = filter.W+ 2 * alpha*filter.P^-1 * conj(X) * r;
    filter.w = filter.F*filter.W;
end
end
```