Adaptive Array Signal Processing

[5SSC0]

Assignment Part 1A: Adaptive Algorithms

REPORT

Group number:

Names including ID:
1: Idan Grady 1912976
2: Yoosef Bidardel 1822535

Date: 12/02/2021

1.2 Scenario 1: Known statistics

1.2.1 Wiener filter

a

Use the

$$W_0 = R^{-1} \cdot r_{ex}$$
$$=$$

$$\begin{bmatrix} 2/3 & 1/3 \\ 1/3 & 2/3 \end{bmatrix} \cdot \begin{bmatrix} 0 & 3 \end{bmatrix}^T = \begin{bmatrix} \frac{3}{3} & \frac{6}{3} \end{bmatrix}^T$$

b

$$\begin{aligned} r[k] &= e[k] - w^T[k]x[k]. \\ r_{xr}[l] &= \\ &\quad E[x[k](e[k-l] - w^T x(k-l))] \\ &\quad = E[x[k]e[k-l]] - E[x[k]w^T x(k-l)]. \\ &\quad = r_{xe} - w^T R_x \end{aligned}$$

Hence, when W_{opt} this would become 0.

c

In order to estimate the statistics for Rx and rex, one would sample a sufficient amount of data from x[k] and e[k]. This sample should be large enough to accurately represent the underlying statistics. To further improve the accuracy of the estimates, it may be necessary to monitor changes in the statistics over time and adjust the sample size as needed. The tradeoff in this case is between accuracy and computational efficiency. As the amount of samples increases, the estimates should converge towards the correct process or signal statistics according to the Strong Law of Large Numbers. However, this process may require a significant amount of computational resources and time.

1.2.2 Steepest Gradient Descent

d

SGD algorithm converges when the error does not change much anymore, indicating that the minimum point of the cost function is being approached. Hence W[k+1] = w[k]

$$w[k+1] = w[k] - 2a(r_{ex} - R_x w_k)$$

$$w[k+1] = w[k] =>$$

$$w[k] = w[k] - 2a(r_{ex} - R_x w_k) = -2a(r_{ex} - R_x w_k) r_{ex} - R_x w_k w_k = R_x^{-1} r_{ex}$$

Also, it is important to mention that we have to factor in stability. To do so, we have to check the stability as well. By Eigenvalue decomposition we have:

$$|1 - 2\alpha\lambda_i| < 1 => 0 < \alpha < \frac{1}{\lambda_{max}}$$

Hence, we can see that once we let SGD converge to the optimal, it would find the same W_{opt} as the Wiener filter.

e

To calculate the range of **adaptation constant** (α) first, we have to derive all the eigenvalues.

$$R_x = \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix}$$

we can do this by solving $det(R_x - \lambda I) = 0$.

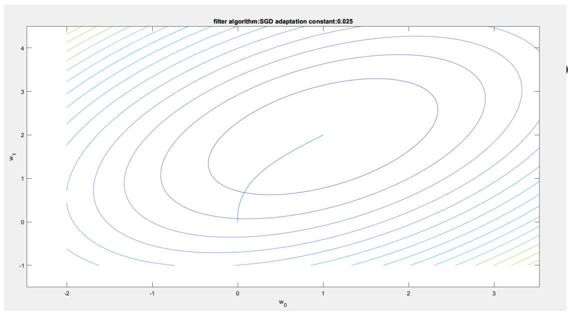
$$\begin{vmatrix} 2-\lambda & -1 \\ -1 & 2-\lambda \end{vmatrix} => (2-\lambda)^2 = 1$$

As a result of which $\lambda_1 = 1$ and $\lambda_2 = 3$. the range of α would be :

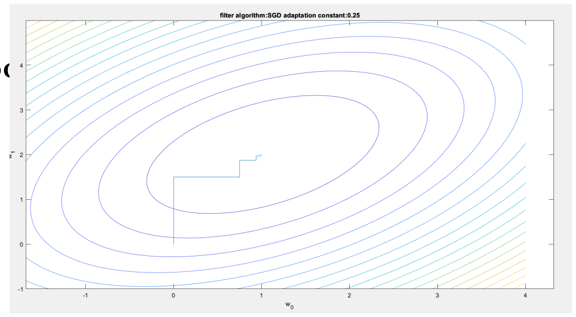
$$0 < \alpha < \frac{1}{3}$$

f

After implementing the SGD part in Matlab, the results are shown below:



(a) $\alpha = 0.025$



(b) $\alpha = 0.25$

Figure 1: Learning curve in contour plot J with 2 different α

Since the $\Gamma_x = \frac{\lambda_{max}}{\lambda_{min}} = 3$ the convergence path is not a straight line. Instead, it converges faster in the y-axis than in the x's.

1.2.3 Newton's Method

g

Newtons method can written as

$$\begin{aligned} w[k+1] &= w[k] + 2aR_x^{-1}(r_{ex} - R_x w[k]] \\ &= w[k] + 2aR_x^{-1}(x[k]e[k] - x[k]x^T[k]w[k]) \end{aligned} \tag{1}$$

$$\begin{aligned} &= w[k] + 2aR_x^{-1}r_{ex} - 2aw[k] \\ &w[k](1-2a) + 2aR_x^{-1}r_{ex} \end{aligned} \tag{2}$$

The main observation here is in equation (1)(before applying the inverse), is a quadratic equation. Which means that the quadratic term influence the curvature, whereas the linear term influence the gradient or the "translation"of along the line. This means, that the process of whitening the auto-correlation of the input signals is equivalent to transforming the input signals into a linear space where they are uncorrelated and have unit variance. Yet we know that only the eigenvalues of the Hassian influence the curvature and convergence rate. Yet the r_{ex} only determines the direction of the gradient and does not determine "how much" the filter weights should be updated.

To conclude: Since the Hessian of the quadratic term is a constant matrix, it has the same eigenvalues for all filter weights. This means that the quadratic term affects the curvature of the objective function equally in all directions, and hence all filter weights converge at the same rate.

h

$$\begin{aligned} d[k+1] &= (1-2a)d[k] \\ &= (1-2a)^2 d[k-1] \\ &...(1-2a)^k d[0] \end{aligned}$$

hence $|(1-2a)| < 1$
$\rightarrow 0 < a < 1$

i

As can be seen, it converges with only one step

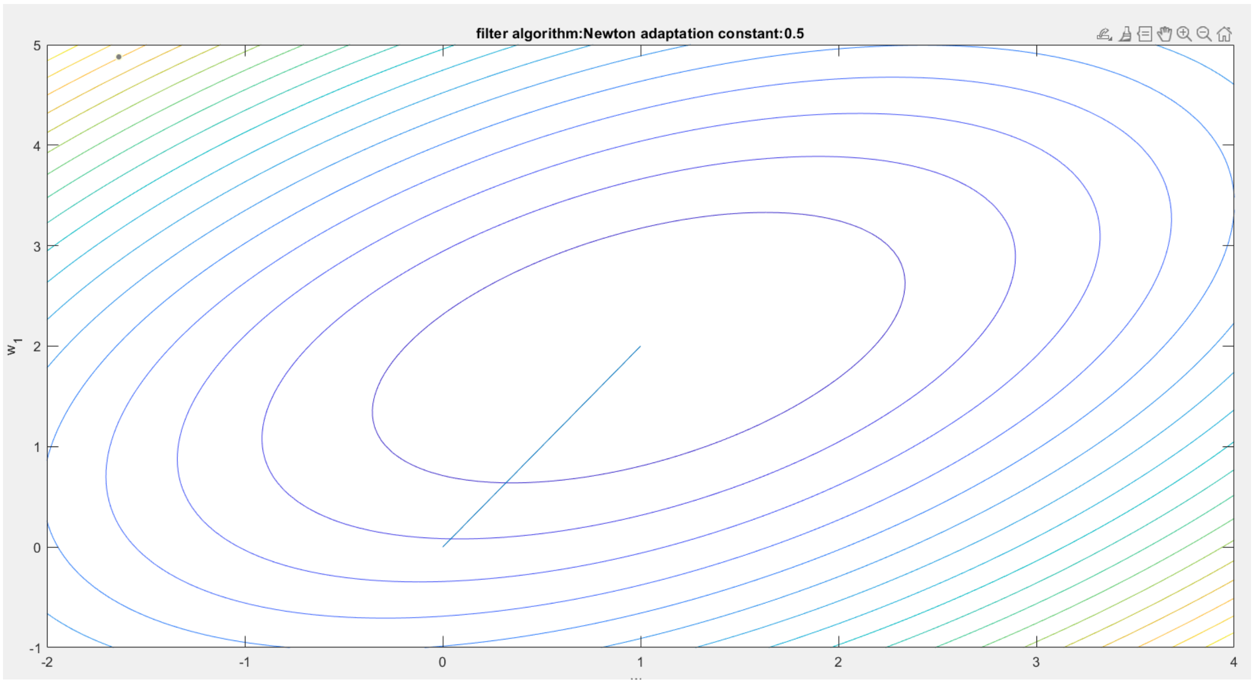
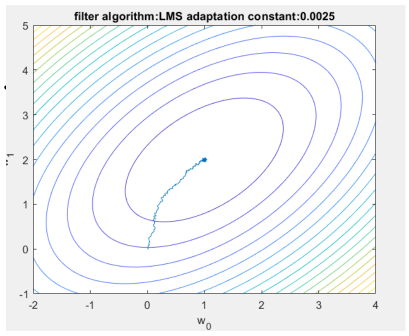


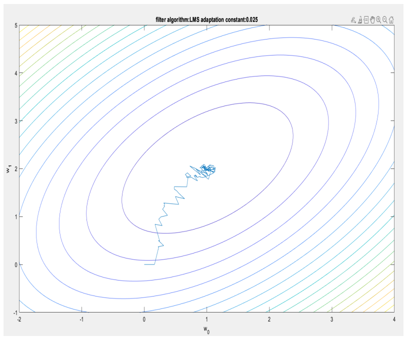
Figure 2: 1-step convergence with Lr$=$ 0.5

1.3 Scenario 2: Unknown statistics

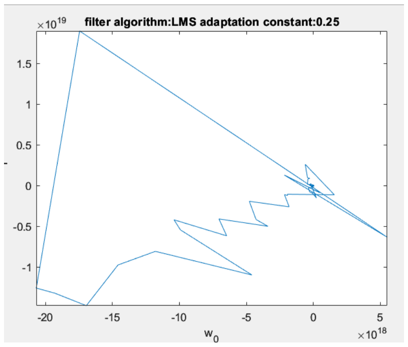
1.3.1 LMS and NLMS



(a) Lr= 0.0025



(b) Lr= 0.025



(c) Lr= 0.25

Figure 3: LMS with 3 different learning rates,

The relevant code is shown below:

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

A smaller α results in a slower convergence rate but may provide more stable performance, while a larger value of α results in a faster convergence rate but may lead to instability or overshoot (diverge).

k

After implementing the normalization factor, we can vividly see that the deviation between the true path and the estimated one decreases.

The NLMS adapts the step size to the power of the eigenvalues. This means, that when the eigenvalues are larger, the gradient is steeper. Hence, when adapting the step size, we mitigate divergence.

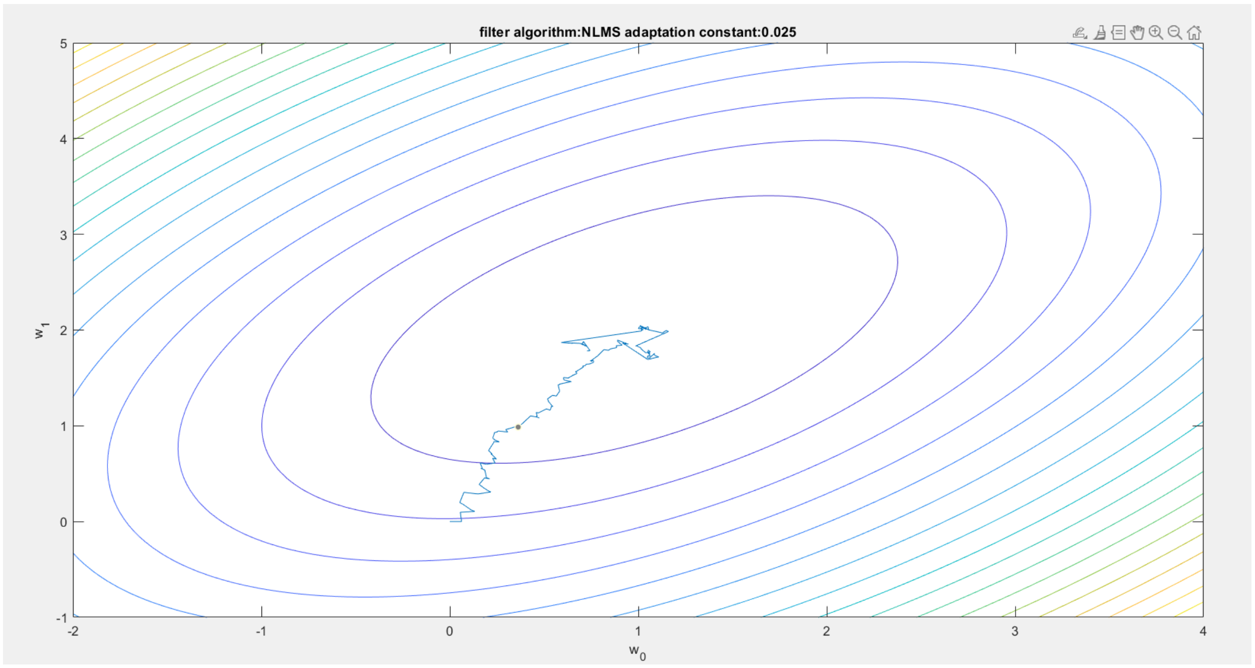


Figure 4: NLMS with learning rate 0.025

```
if strcmpi(filter_type,'NLMS')
    %implement the NLMS update rule here
    alpha=filter.adaptation_constant;
    [n,d]= size(x);
    filter.w= w_old + 2/(dot(x,x')/d)*alpha* x*r;
end
```

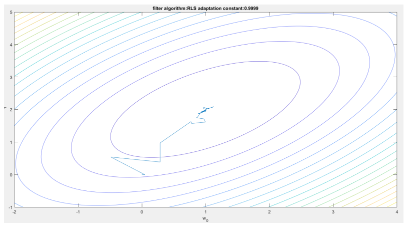
1.3.2 RLS and FDAF

c

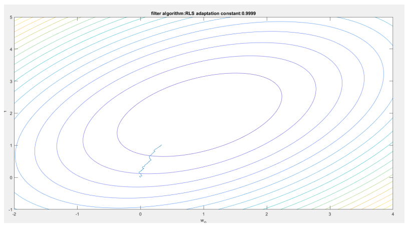
There are some advantages to deploy RLS and ADAF methods instead of previous ones. When it comes to RLS and FADAF we try to estimate parameters by recursion and having present and past data(history) while on the contrary other methods try to minimize the cost function considering the input data.

More specifically, the RLS can come in handy when we are dealing with time-varying or non-stationary data since it is likely to adapt to rapid changes. The FADAF on the other hand, can solve the complexity problem by reducing the calculation parameters in each iteration.

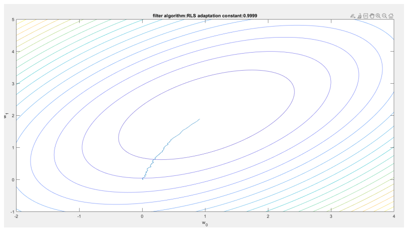
Some of the mentioned methods such as SGD take the state of parameters for granted, meaning that we assume R_x is provided while in practice this is not the case. This applies to other methods as well such as Newton because not only it needs the R_x but it is also computationally expensive to have the inverse matrix.



(a) 200 samples. R = eye(2)



(b) 200 samples R = eye(2)*0.02



(c) 2000 samples, R = eye(2)*0.02

Figure 5: RLS with $1 * 10^{-4}$ LR,

e

```
if strcmpi(filter_type,'FDAF')
   alpha = filter.adaptation_constant;
   X = filter.F * x;
   beta = 0.5;
   filter.est_p = beta * filter.est_p + (1 - beta) * (X .*
      conj(X)')/length(x);
   %convert back to frequency domain
   W_old = filter.F_inverse * w_old;

   % Precompute diag_spectrum and inv_diag_spectrum
   diag_spectrum = diag(filter.est_p);
   inv_diag_spectrum = inv(diag(diag_spectrum));

   % Update w in frequency domain
   W_new  = W_old + (2 * alpha) * inv_diag_spectrum * conj(X)
      * r;
   %return to time domain
   filter.w = filter.F * W_new;
end
```

f

To compare these 4 methods we have to take a look at each of their advantages:

1) **FADAF** Since it uses a finite difference adaptive filter, it is faster and more accurate than RLS with the complexity scale of O(N^2).

2) **RLS** this method is beneficial as the adaptability rate to new input is high with the complexity scale of O(N^3).

3) **LMS** most computationally efficient method among others with complexity scale of O(N). Nevertheless, it requires more time to reach the same level of accuracy than other methods(it needs more iteration to converge).

4) **NLMS** the normalized version of LMS with the same complexity. Although it is faster than FADAF and RLS, it is less accurate.

   the ranking of these methods largely depends on the application we are dealing with. Let's say accuracy is more important, then we should definitely go for RLS(**best**), and at the bottom would be LMS and NLMS (**worst**). Other than that if we want both high accuracy and less computational power, trade of is our only choice which leaves us with NLSM and FADAF methods, with the latter being faster than RLS and the former being less accurate than RLS or FDAF.

Appendices

update$_{filter}$

```
 function [ filter ] = update_filter( filter,e )

x=filter.x_delayed;
w_old = filter.w;
R_old = filter.R;
r=filter.r;
re_old = filter.re;
filter_type=filter.type;

%% A1 scenario 1:f
if strcmpi(filter_type,'SGD')
    %implement the SGD update rule here
    alpha=filter.adaptation_constant;
    Rx =[2 -1; -1 2];
    rex = [0;3];
    filter.w= w_old+2*alpha*(rex- Rx*w_old);
end

 %% A1 scenario 1:i
 if strcmpi(filter_type,'Newton')
    alpha=filter.adaptation_constant;
    Rx =[2 -1; -1 2];
    rex = [0;3];
    t=(rex-Rx*w_old);
    filter.w= w_old + 2*alpha*Rx\(rex-Rx*w_old);

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

%% A1 scenario 2:b
if strcmpi(filter_type,'NLMS')
    %implement the NLMS update rule here
    alpha=filter.adaptation_constant;
    [n,d]= size(x);
    filter.w= w_old + 2/(dot(x,x')/d)*alpha* x*r;
end
```

```
%% A1 scenario 2:d
if strcmpi(filter_type,'RLS')
    %implement the RLS update rule here
    lambda=filter.adaptation_constant;
    disp(size(filter.R))
    disp(lambda)

    down =x'*R_old*x + lambda;
    up = R_old*x;
    g=(up)/( down);
    filter.R =lambda^(-2)*(R_old - g*x'*R_old);
    filter.re = lambda^2* re_old + x*e;
    filter.w= filter.R* filter.re;
end

 %% A1 scenario 2:e
 if strcmpi(filter_type,'FDAF')
    alpha = filter.adaptation_constant;
    X = filter.F * x;
    beta = 0.5;
    filter.est_p = beta * filter.est_p + (1 - beta) * (X .*
       conj(X)')/length(x);
    %convert back to frequency domain
    W_old = filter.F_inverse * w_old;

    % Precompute diag_spectrum and inv_diag_spectrum
    diag_spectrum = diag(filter.est_p);
    inv_diag_spectrum = inv(diag(diag_spectrum));

    % Update w in frequency domain
    W_new  = W_old + (2 * alpha) * inv_diag_spectrum * conj(X)
       * r;
    %return to time domain
    filter.w = filter.F * W_new;
 end
end
```