Adaptive Array Signal Processing

[5SSC0]

Assignment Part 1A: Adaptive Algorithms

REPORT

Group number: 3

Names & ID:
1: Floris Naber 1253514
2: Shao Hsuan Hung 1723219

Date: 12/02/2021

1.2 Scenario 1: Known statistics

1.2.1 Wiener filter

a

First we write out $E\{r^2[k]\}$:

$$E\{r^2[k]\} = E\{(e[k] - \hat{e}[k])^2\} = E\{(e[k] - \underline{\mathbf{w}}^\top \underline{\mathbf{x}}[k])(e[k] - \underline{\mathbf{x}}^\top[k]\underline{\mathbf{w}})\}. \tag{1}$$

Now we expand the square and take out $\underline{\mathbf{w}}$ of the $E\{\cdot\}$ operator:

$$\begin{aligned} E\{r^2[k]\} &= E\{e^2[k] - \underline{\mathbf{w}}^\top \underline{\mathbf{x}}[k]e[k] - e[k]\underline{\mathbf{x}}^\top[k]\underline{\mathbf{w}} + \underline{\mathbf{w}}^\top \underline{\mathbf{x}}[k]\underline{\mathbf{x}}^\top[k]\underline{\mathbf{w}}\} \\ &= E\{e^2[k]\} - \underline{\mathbf{w}}^\top E\{\underline{\mathbf{x}}[k]e[k]\} - E\{e[k]\underline{\mathbf{x}}^\top[k]\}\underline{\mathbf{w}} + \underline{\mathbf{w}}^\top E\{\underline{\mathbf{x}}[k]\underline{\mathbf{x}}^\top[k]\}\underline{\mathbf{w}}. \end{aligned} \tag{2}$$

By using the definition of correlation we can write out the above equation as:

$$E\{r^2[k]\} = r_e - \underline{\mathbf{w}}^\top \underline{\mathbf{r}}_{ex} - \underline{\mathbf{r}}_{ex}^\top \underline{\mathbf{w}} + \underline{\mathbf{w}}^\top \mathbf{R}_x \underline{\mathbf{w}}, \tag{3}$$

where $r_e = E\{e^2[k]\}$ is the autocorrelation of $e[k]$, $\underline{\mathbf{r}}_{ex} = E\{e[k]\underline{\mathbf{x}}[k]\}$ is the correlation between $e[k]$ and $\underline{\mathbf{x}}[k]$ and $\mathbf{R}_x = E\{\underline{\mathbf{x}}[k]\underline{\mathbf{x}}^\top[k]\}$ is the autocorrelation of $\underline{\mathbf{x}}[k]$. Assuming $\mathbf{R}_x$ is positive definite, then this is a convex function with respect to $\underline{\mathbf{w}}$. We can find the optimal point $\underline{\mathbf{w}}_o$, where the gradient with respect to $\underline{\mathbf{w}}$ of the cost function is zero:

$$\begin{aligned} \nabla J(\underline{\mathbf{w}}) &= 2\mathbf{R}_x \underline{\mathbf{w}} - 2\underline{\mathbf{r}}_{ex} \\ 0 = \nabla J(\underline{\mathbf{w}}_o) &= 2\mathbf{R}_x \underline{\mathbf{w}}_o - 2\underline{\mathbf{r}}_{ex} \quad \rightarrow \quad \underline{\mathbf{w}}_o = \mathbf{R}_x^{-1}\underline{\mathbf{r}}_{ex}. \end{aligned} \tag{4}$$

Finally we can calculate the optimal filter weight $\underline{\mathbf{w}}_o$:

$$\underline{\mathbf{w}}_o = \mathbf{R}_x^{-1}\underline{\mathbf{r}}_{ex} = \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix}^{-1} \begin{bmatrix} 0 \\ 3 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}. \tag{5}$$

b

The correlation between the input $\underline{\mathbf{x}}[k]$ and the residual $r[k]$, namely $\underline{\mathbf{r}}_{\mathbf{xr}}$, is defined by:

$$\underline{\mathbf{r}}_{\mathbf{xr}} = E\{\underline{\mathbf{x}}[k]r[k]^\top\}. \tag{6}$$

Now we can expand and rewrite $r[k]$ in terms of $e[k]$, $\underline{\mathbf{x}}[k]$ and $\underline{\mathbf{w}}$:

$$\begin{aligned} \underline{\mathbf{r}}_{\mathbf{xr}} &= E\{\underline{\mathbf{x}}[k](e[k] - \hat{e}[k])^\top\} \\ &= E\{\underline{\mathbf{x}}[k](e[k] - \underline{\mathbf{w}}^\top \underline{\mathbf{x}}[k])^\top\} \\ &= E\{\underline{\mathbf{x}}[k](e[k] - \underline{\mathbf{x}}^\top[k]\underline{\mathbf{w}})\}. \end{aligned} \tag{7}$$

We now expand further and substitute in $\mathbf{R}_x$ and $\underline{\mathbf{r}}_{ex}$:

$$\begin{aligned}\underline{\mathbf{r}}_{\mathbf{xr}} &= E\{\underline{\mathbf{x}}[k]e[k] - \underline{\mathbf{x}}[k]\underline{\mathbf{x}}^{\top}[k]\underline{\mathbf{w}}\} \\ &= E\{\underline{\mathbf{x}}[k]e[k]\} - E\{\underline{\mathbf{x}}[k]\underline{\mathbf{x}}^{\top}[k]\}\underline{\mathbf{w}} \\ &= \underline{\mathbf{r}}_{ex} - \mathbf{R}_x\underline{\mathbf{w}}.\end{aligned} \tag{8}$$

If the weights $\underline{\mathbf{w}}$ are equal to the optimal weight for the Wiener filter $\underline{\mathbf{w}}_o$, then we can substitute in the solution found in equation 4 and simplify:

$$\begin{aligned}\underline{\mathbf{r}}_{\mathbf{xr}} &= \underline{\mathbf{r}}_{ex} - \mathbf{R}_x\underline{\mathbf{w}}_o \\ &= \underline{\mathbf{r}}_{ex} - \mathbf{R}_x\mathbf{R}_x^{-1}\underline{\mathbf{r}}_{ex} \\ &= \underline{\mathbf{r}}_{ex} - \underline{\mathbf{r}}_{ex} \\ &= 0.\end{aligned} \tag{9}$$

This result shows that the correlation between the input $\underline{\mathbf{x}}[k]$ and the residual $r[k]$, $\underline{\mathbf{r}}_{\mathbf{xr}}$, is zero, when the weights are optimal. This means that they are uncorrelated, which is the goal of the filter.

c

If the statistical information $\mathbf{R}_x$ and $\underline{\mathbf{r}}_{ex}$ are unavailable, then we can estimate them from the observed input signal $\underline{\mathbf{x}}[k]$ and observed reference signal $e[k]$ by using time averaging:

$$\mathbf{R}_x \approx \hat{\mathbf{R}}_x = \frac{1}{L}\sum_{i=0}^{L-1}\tilde{\mathbf{R}}_x[k-i] = \frac{1}{L}\sum_{i=0}^{L-1}\underline{\mathbf{x}}[k-i]\underline{\mathbf{x}}^{\top}[k-i] \tag{10}$$

$$\mathbf{r}_{ex} \approx \hat{\mathbf{r}}_{ex} = \frac{1}{L}\sum_{i=0}^{L-1}\tilde{\underline{\mathbf{r}}}_{ex}[k-i] = \frac{1}{L}\sum_{i=0}^{L-1}\underline{\mathbf{x}}[k-i]e[k-i], \tag{11}$$

where $\tilde{\mathbf{R}}_x[k]$ and $\tilde{\underline{\mathbf{r}}}_{ex}[k]$ are the "instantaneous" estimated statistics at a certain discrete time instant k. We average these instantaneous estimated statistics over a period of length L to improve the estimate and make it more stable. For an application you would use a running average, as the statistics can change over time depending on a change in the input and/or change in the environment. The trade-off is in choosing L. If you choose L to be very large, then the statistics will be estimated with a lot of data, and the estimate is very accurate if the statistics do not change over time. On the other hand, if the statistics do change over time, for example due to a change in input or change in the environment, then a large L means that the estimated statistics update very slowly, which can make them inaccurate right after such a change.

1.2.2 Steepest Gradient Descent

d

Based on the steepest gradient descent principle, the parameters are always update in negative gradient direction. The update rule is:

$$\underline{\mathbf{w}}[k+1] \doteq \underline{\mathbf{w}}[k] - \alpha\underline{\nabla}, \tag{12}$$

where adaptation constant $\alpha > 0$, $\underline{\nabla} = -2(\underline{\mathbf{r}}_{ex} - \mathbf{R}_x\underline{\mathbf{w}}[k])$ is the gradient of the cost function with respect to $\underline{\mathbf{w}}[k]$. When the SGD algorithm reaches steady state (thus for large k), the value of $\underline{\mathbf{w}}[k]$ will stabilize:

$$\lim_{k\to\infty} \underline{\mathbf{w}}[k+1] \simeq \underline{\mathbf{w}}[k]. \tag{13}$$

If we fill this result into the update rule, we get:

$$\lim_{k\to\infty} \underline{\mathbf{w}}[k] \simeq \underline{\mathbf{w}}[k] + 2\alpha(\underline{\mathbf{r}}_{ex} - \mathbf{R}_x\underline{\mathbf{w}}[k]) \tag{14}$$

Thus we have that:

$$\lim_{k\to\infty} 2\alpha(\underline{\mathbf{r}}_{ex} - \mathbf{R}_x\underline{\mathbf{w}}[k]) \simeq 0 \tag{15}$$

From this it can be seen that $\underline{\mathbf{w}}[k] \simeq \mathbf{R}_x^{-1}\underline{\mathbf{r}}_{ex}$ for large k. To conclude, the SGD algorithm will converge to the Wiener solution:

$$\lim_{k\to\infty} \underline{\mathbf{w}}[k] \simeq \mathbf{R}_x^{-1}\underline{\mathbf{r}}_{ex} = \underline{\mathbf{w}}_o. \tag{16}$$

e

To do a stability analysis, first we define a vector $\underline{\mathbf{d}}[k]$ that describes the difference between a vector of weights at time instance k, namely $\underline{\mathbf{w}}[k]$, and the optimal Wiener filter weights $\underline{\mathbf{w}}_o$ as follows:

$$\underline{\mathbf{d}}[k] = \underline{\mathbf{w}}[k] - \underline{\mathbf{w}}_o. \tag{17}$$

Now we use this definition to remove the weights $\underline{\mathbf{w}}[k]$ from the weight update equation (12), by using substitution and then simplifying:

$$\begin{aligned}
\underline{\mathbf{w}}[k+1] &= \underline{\mathbf{w}}[k] + 2\alpha(\underline{\mathbf{r}}_{ex} - \mathbf{R}_x\underline{\mathbf{w}}[k]) \\
\underline{\mathbf{w}}[k+1] - \underline{\mathbf{w}}_o &= \underline{\mathbf{w}}[k] - \underline{\mathbf{w}}_o + 2\alpha\big(\underline{\mathbf{r}}_{ex} - \mathbf{R}_x(\underline{\mathbf{d}}[k] + \underline{\mathbf{w}}_o)\big) \\
\underline{\mathbf{d}}[k+1] &= \underline{\mathbf{d}}[k] + 2\alpha\big(\underline{\mathbf{r}}_{ex} - \mathbf{R}_x(\underline{\mathbf{d}}[k] + \mathbf{R}_x^{-1}\underline{\mathbf{r}}_{ex})\big) \\
\underline{\mathbf{d}}[k+1] &= \underline{\mathbf{d}}[k] + 2\alpha\big(\underline{\mathbf{r}}_{ex} - \mathbf{R}_x\underline{\mathbf{d}}[k] - \mathbf{R}_x\mathbf{R}_x^{-1}\underline{\mathbf{r}}_{ex}\big) \\
\underline{\mathbf{d}}[k+1] &= \underline{\mathbf{d}}[k] - 2\alpha\mathbf{R}_x\underline{\mathbf{d}}[k] \\
\underline{\mathbf{d}}[k+1] &= (I - 2\alpha\mathbf{R}_x)\underline{\mathbf{d}}[k],
\end{aligned} \tag{18}$$

where I is the identity matrix with the same dimensions as the auto-correlation matrix $\mathbf{R}_x$. This recursive function of the difference vector $\underline{\mathbf{d}}[k]$ can be written out in terms of the starting difference $\underline{\mathbf{d}}[0]$:

$$\begin{aligned}
\underline{\mathbf{d}}[k] &= (I - 2\alpha\mathbf{R}_x)\underline{d}[k-1] \\
\underline{\mathbf{d}}[k] &= (I - 2\alpha\mathbf{R}_x)(I - 2\alpha\mathbf{R}_x)\underline{\mathbf{d}}[k-2] \\
\underline{\mathbf{d}}[k] &= (I - 2\alpha\mathbf{R}_x)^2\underline{\mathbf{d}}[k-2] \\
\underline{\mathbf{d}}[k] &= (I - 2\alpha\mathbf{R}_x)^k\underline{\mathbf{d}}[0].
\end{aligned} \tag{19}$$

For convergence we know that weights $\underline{\mathbf{w}}[k]$ converge to the Wiener filter solution and that the weights do not change in steady state. Thus we have that:

$$\lim_{k\to\infty} \underline{\mathbf{d}}[k] \simeq \underline{\mathbf{0}}, \tag{20}$$

where $\underline{\mathbf{0}}$ is the zero vector. That means the α should maintain the following constraint to make the SGD algorithm stable:

$$\lim_{k\to\infty} (I - 2\alpha R_x)^k = 0. \tag{21}$$

Then we do the eigenvalue decomposition on the auto-correlation $\mathbf{R}_x$:

$$(I - 2\alpha\mathbf{R}_x)^k = (\mathbf{Q}\mathbf{Q}^h - 2\alpha\mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^h)^k = \mathbf{Q}(I - 2\alpha\mathbf{\Lambda})^k\mathbf{Q}^h, \tag{22}$$

where Q is a matrix containing the eigenvectors of $\mathbf{R}_x$ and Λ is a diagonal matrix containing the eigenvalues of $\mathbf{R}_x$. Now we substitute this result in equation 18:

$$\begin{aligned}
\underline{\mathbf{d}}[k] &= (I - 2\alpha\mathbf{R}_x)^k\underline{\mathbf{d}}[0] \\
\underline{\mathbf{d}}[k] &= \mathbf{Q}(I - 2\alpha\mathbf{\Lambda})^k\mathbf{Q}^h\underline{\mathbf{d}}[0] \\
\mathbf{Q}^h\underline{\mathbf{d}}[k] &= (I - 2\alpha\mathbf{\Lambda})^k\mathbf{Q}^h\underline{\mathbf{d}}[0] \\
\underline{\mathbf{D}}[k] &= (I - 2\alpha\mathbf{\Lambda})^k\underline{\mathbf{D}}[0],
\end{aligned} \tag{23}$$

where $\underline{\mathbf{D}}[k]$ is $\underline{\mathbf{d}}[k]$ that underwent a matrix transformation according to matrix $\mathbf{Q}^h$. Thus we have that

$$\lim_{k\to\infty} \underline{\mathbf{d}}[k] \simeq \underline{\mathbf{0}} \Rightarrow \lim_{k\to\infty} \underline{\mathbf{D}}[k] \simeq \underline{\mathbf{0}}. \tag{24}$$

Thus the convergence of the SGD algorithm is stable if and only if:

$$\lim_{k\to\infty} (I - 2\alpha\mathbf{\Lambda})^k = 0. \tag{25}$$

Since both I and Λ are diagonal, we know that the individual elements of the matrix $(I-2\alpha\mathbf{\Lambda})$ are multiplied only with themselves on every iteration, which should converge to 0. Thus the absolute value of these individual elements should be smaller than 1 for convergence. Thus finally we can calculate values for α that lead to stable convergence:

$$\begin{aligned}
&\lim_{k\to\infty} (1 - 2\alpha\lambda_i)^k = 0 \\
&\Rightarrow |1 - 2\alpha\lambda_i| < 1 \\
&\Rightarrow 0 < \alpha < \frac{1}{\lambda_{\max}}.
\end{aligned} \tag{26}$$

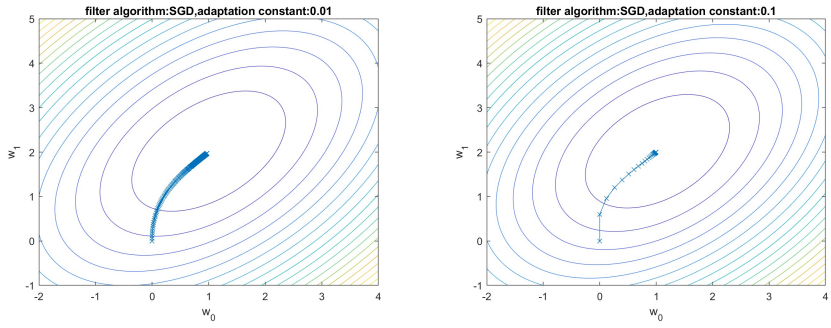
Where λ_i indicates the i'th eigenvalue of the auto-correlation matrix $\mathbf{R}_x$ and $\lambda_{\max}$ indicates the largest of those eigenvalues. Thus now we calculate the set of eigenvalues λ of $\mathbf{R}_x$ according to the following equation:

$$\begin{aligned} 0 &= \det\big(\mathbf{R}_x - \lambda I\big) = \begin{vmatrix} 2-\lambda & -1 \\ -1 & 2-\lambda \end{vmatrix} \\ 0 &= (2-\lambda)^2 - 1 = \lambda^2 - 4\lambda + 3 = (\lambda - 1)(\lambda - 3) \\ &\Rightarrow \lambda = \{1, 3\}. \end{aligned} \tag{27}$$

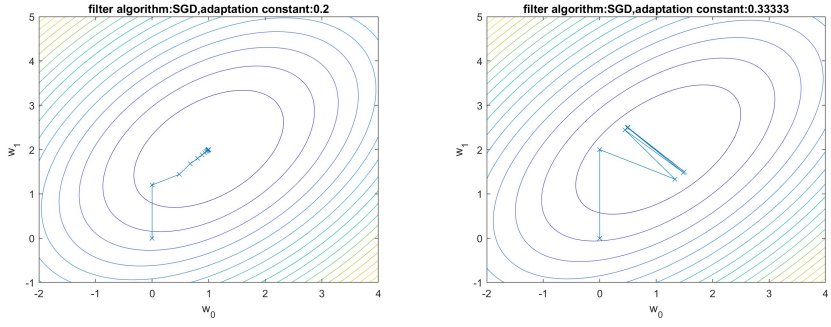
We conclude from this that the largest eigenvalue of $\mathbf{R}_x$ is described by $\lambda_{\max} = 3$. From this it can be concluded that the convergence of the SGD algorithm is stable if and only if $0 < \alpha < \frac{1}{3}$.

f

The SGD algorithm for different α can be seen in figure 1. It can be seen that for $\alpha = \frac{1}{3}$ the convergence is not stable, as this is equal to the inverse of the largest eigenvalue of $\mathbf{R}_x$. For smaller values of α the weights converge more smoothly, but slower.



(a) $\alpha = 0.01$ (b) $\alpha = 0.1$



(c) $\alpha = 0.2$ (d) $\alpha = \frac{1}{3}$

Figure 1: SGD with different adaptation constants α.

1.2.3 Newton's Method

g

For SGD it can be seen from equation 18 that the change of every element in $\underline{\mathbf{d}}[k]$ depends on the elements in $\mathbf{R}_x$ and thus update at different rates if the eigenvalues of $\mathbf{R}_x$ are different. For Newton's update rule we have that:

$$\begin{aligned}
\underline{\mathbf{w}}[k+1] &= \underline{\mathbf{w}}[k] + 2\alpha\mathbf{R}_x^{-1}(\underline{\mathbf{r}}_{ex} - \mathbf{R}_x\underline{\mathbf{w}}[k]) \\
\underline{\mathbf{w}}[k+1] - \underline{\mathbf{w}}_o &= \underline{\mathbf{w}}[k] - \underline{\mathbf{w}}_o + 2\alpha\mathbf{R}_x^{-1}(\underline{\mathbf{r}}_{ex} - \mathbf{R}_x\underline{\mathbf{w}}[k]) \\
\underline{\mathbf{d}}[k+1] &= \underline{\mathbf{d}}[k] + 2\alpha\mathbf{R}_x^{-1}\big(\underline{\mathbf{r}}_{ex} - \mathbf{R}_x(\underline{\mathbf{d}}[k] - \underline{\mathbf{w}}_o)\big) \\
\underline{\mathbf{d}}[k+1] &= \underline{\mathbf{d}}[k] + 2\alpha\mathbf{R}_x^{-1}\big(\underline{\mathbf{r}}_{ex} - \mathbf{R}_x(\underline{\mathbf{d}}[k] - \mathbf{R}_x^{-1}\underline{\mathbf{r}}_{ex})\big) \\
\underline{\mathbf{d}}[k+1] &= \underline{\mathbf{d}}[k] + 2\alpha\mathbf{R}_x^{-1}\big(\underline{\mathbf{r}}_{ex} - \mathbf{R}_x\underline{\mathbf{d}}[k] - \mathbf{R}_x\mathbf{R}_x^{-1}\underline{\mathbf{r}}_{ex}\big) \\
\underline{\mathbf{d}}[k+1] &= \underline{\mathbf{d}}[k] - 2\alpha\mathbf{R}_x^{-1}\mathbf{R}_x\underline{\mathbf{d}}[k] \\
\underline{\mathbf{d}}[k+1] &= \underline{\mathbf{d}}[k] - 2\alpha\underline{\mathbf{d}}[k] \\
\underline{\mathbf{d}}[k+1] &= (1-2\alpha)\underline{\mathbf{d}}[k],
\end{aligned} \tag{28}$$

where $\underline{\mathbf{d}}[k] = \underline{\mathbf{w}}[k] - \underline{\mathbf{w}}_o$, and $\underline{\mathbf{w}}_o = \mathbf{R}_x^{-1}\underline{\mathbf{r}}_{ex}$, the optimal weights for a Wiener filter. It can be seen that the difference $\underline{\mathbf{d}}[k]$ between the current weights and the optimal weights $\underline{\mathbf{w}}_o$ is dependent only on the previous weights and α, and not on the input $\underline{\mathbf{x}}[k]$ or its statistics. Every element in $\underline{\mathbf{d}}[k]$ updates with the same factor $1-2\alpha$, concluding that all filter weights update at the same rate.

h

For convergence we know that as k goes to infinity the difference between the weights and the optimal weights should converge to 0. The update rule for the difference $\underline{\mathbf{d}}[k]$ is recursive and can be written out:

$$\begin{aligned}
\underline{\mathbf{d}}[k] &= (1-2\alpha)\underline{\mathbf{d}}[k-1] \\
\underline{\mathbf{d}}[k] &= (1-2\alpha)(1-2\alpha)\underline{\mathbf{d}}[k-2] \\
\underline{\mathbf{d}}[k] &= (1-2\alpha)^2\underline{\mathbf{d}}[k-2] \\
\underline{\mathbf{d}}[k] &= (1-2\alpha)^k\underline{\mathbf{d}}[0].
\end{aligned} \tag{29}$$

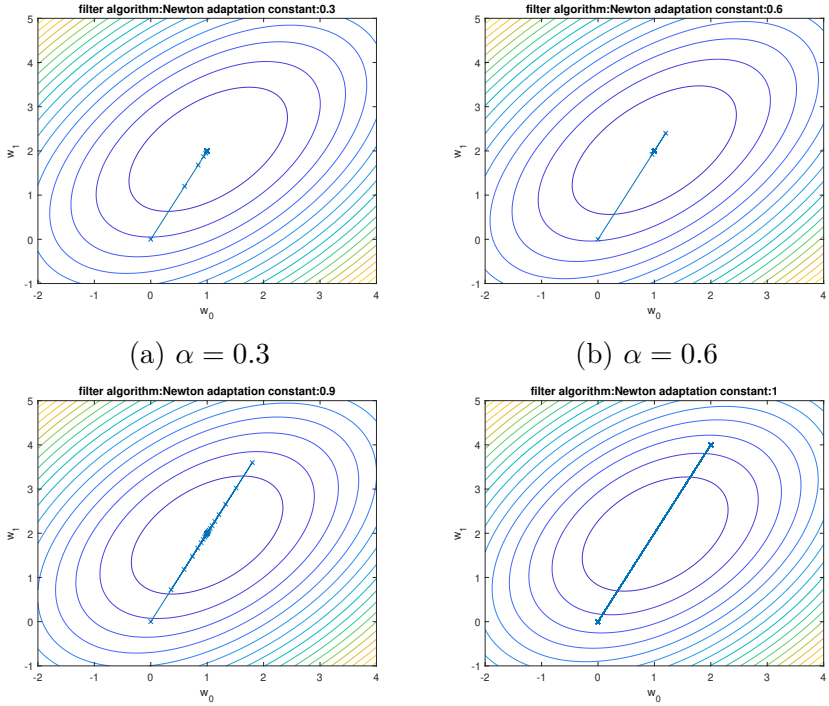
It can be seen from this equation that, as k becomes very large, $\underline{\mathbf{d}}[k]$ only approaches 0, if and only if $(1-2\alpha)^k$ approaches 0, for a random starting point $\underline{\mathbf{d}}[0]$ that is not exactly 0. Thus:

$$\lim_{k\to\infty}(1-2\alpha)^k = 0. \tag{30}$$

This holds if $-1 < (1-2\alpha) < 1$. From this it can be concluded that the convergence process is stable if and only if $0 < \alpha < 1$.

i

It can be seen from figure 2 that for values of α within the proper range the weights converge properly. If the weights are chosen to be relatively large they still overshoot the optimum, but do converge. The weights converge in a straight line to the optimum, confirming that all weights converge at the same rate.



(a) $\alpha = 0.3$ (b) $\alpha = 0.6$

(c) $\alpha = 0.9$ (d) $\alpha = 1$

Figure 2: Newtons method with different adaptation constants α.

1.3 Scenario 2: Unknown statistics

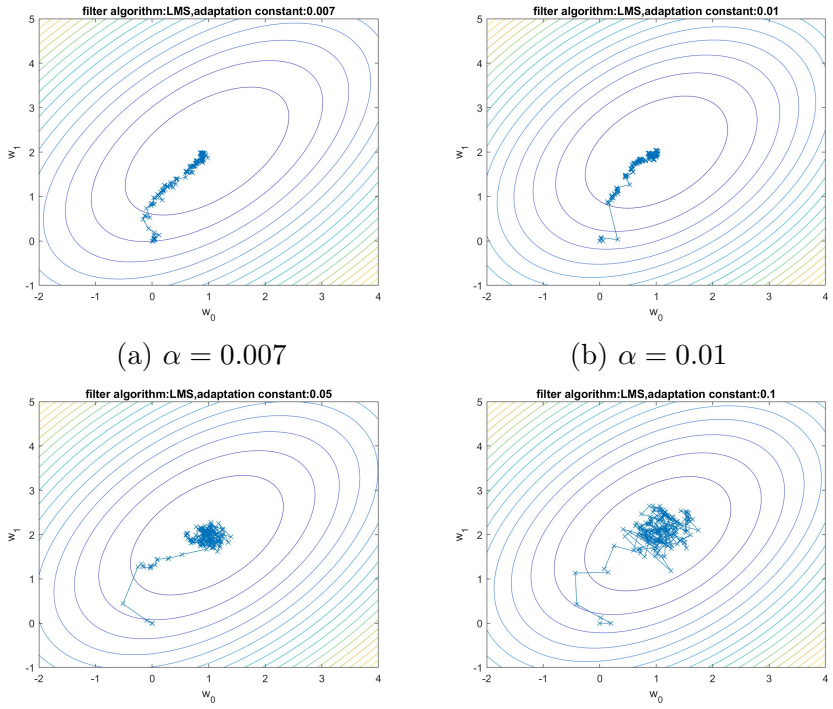
1.3.1 LMS and NLMS

a

Below is the code that updates the weights according to the LMS algorithm.

```
if strcmpi(filter_type,'LMS')
    alpha=filter.adaptation_constant;
    filter.w=w_old+2*alpha*x*r;
end
```

The LMS algorithm for different value of α is shown in figure 3. It can be seen that for $\alpha = 0.1$, the convergence is not stable and not so accurate. For $\alpha = 0.01$ or smaller, the update path of the weights is smoother and reaches a more accurate optimum, but the steps are small and thus it takes more iterations to converge. To sum up, the trade-off for choosing a suitable adaptation constant α is that choosing a larger α makes the weights converge faster, but less accurate convergence. For smaller α, The convergence is more accurate, but slower.



(a) $\alpha = 0.007$ (b) $\alpha = 0.01$

(c) $\alpha = 0.05$ (d) $\alpha = 0.1$

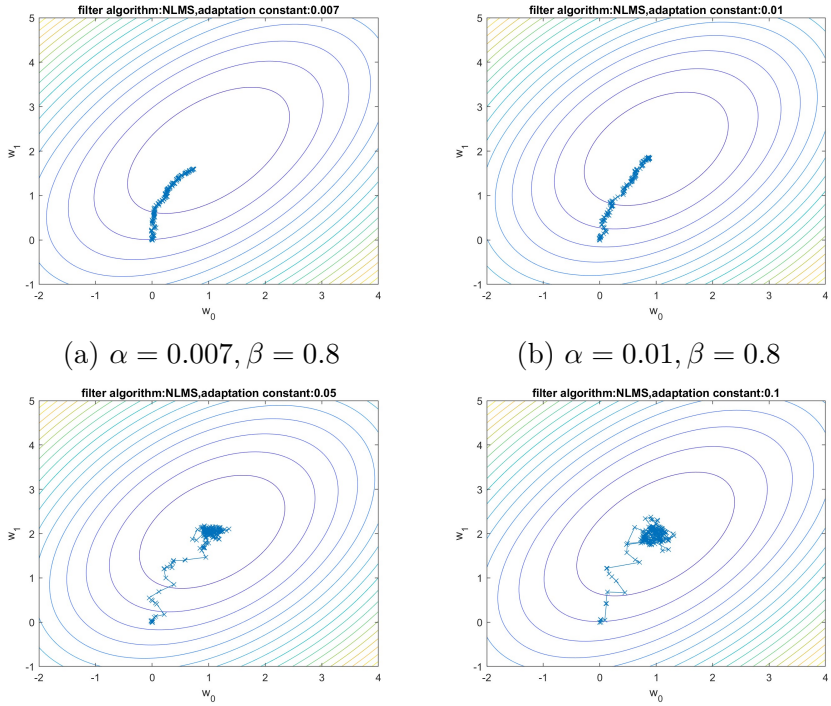
Figure 3: Least mean squares method with different adaptations constants α.

b

Below is the code that updates the weights according to the NLMS algorithm. The only difference with LMS is the normalization factor `var_est`, which is the estimated variance of the input signal $\underline{\mathbf{x}}[k]$ and which is initialized as 1 (no normalization) and updates on every iteration.

```
if strcmpi(filter_type,'NLMS')
    alpha=filter.adaptation_constant;
    beta = 0.8;
    filter.var_est = beta*filter.var_est+(1-beta)*(x'*x)/length(x);
    filter.w=w_old+2*alpha/filter.var_est*x*r;
end
```

The NLMS algorithm for different adaptation constants α is shown in figure 4. When comparing the figure to figure 3 (LMS), it can be seen that the normalization factor makes the weights update more stable (step sizes are more constant), than without normalization. The influence of the normalization factor is that the size of the update steps are normalized, according to the signal power. This makes sure that the convergence behavior does not change when the input power varies over time.



(a) $\alpha = 0.007, \beta = 0.8$ (b) $\alpha = 0.01, \beta = 0.8$

(c) $\alpha = 0.05, \beta = 0.8$ (d) $\alpha = 0.1, \beta = 0.8$

Figure 4: Normalized least mean squares method with different adaptations constants α and constant β.

1.3.2 RLS and FDAF

c

RLS is related to the least squares (LS) algorithm. LS approximates the Wiener filter by making an estimation of the signal statistics and then computing the optimal weights according to the calculation used for the Wiener filter, but with the approximated statistics. The problem with LS is that you have to approximate the signal statistics $\underline{\mathbf{r}}_{ex}$ and $\mathbf{R}_x$ by time-averaging, which can be either inaccurate or slow to adapt (see section c). Also to update the weights, the estimated auto-correlation $\mathbf{R}_x$ needs to be inverted, which is computationally expensive. RLS solves this by estimating $\underline{\mathbf{r}}_{ex}$ and $\mathbf{R}_x^{-1}$ recursively from data. The forgetting factor makes for a better approximation than just time-averaging, as the recent statistics

weight more. Estimating $\mathbf{R}_x^{-1}$ directly means no matrix inversion is needed to update the weights, which makes RLS computationally efficient compared to LS.

FDAF is an alternative to (N)LMS and Newton. It does the same updates of the weights as NLMS, but in frequency domain. Just like NLMS, the update steps are normalized by the power of the input, but now in frequency domain. NLMS suffers from a sub-optimal convergence, because in time domain inputs are often highly auto-correlated and thus the filter weight updates are correlated as well, just as in SGD. FDAF solves this, because in frequency domain the frequency bins are largely uncorrelated for long filters. This makes the filter weights more independent from each other as these weights correspond to different frequencies. As a result the convergence is more similar to Newton, without relying on the signal statistics. NLMS can be computationally expensive for long filters, because of the convolution of the inputs and the weights. In FDAF this convolution is replaced by a multiplication (as we are now in frequency domain), which is cheaper.

d

Below is the code of the RLS algorithm, where `Rx_inv_est` and `rex_est` are the estimation of the inverted auto-correlation matrix $\mathbf{R}_x^{-1}$ and the correlation vector $\underline{\mathbf{r}}_{ex}$. `gamma` is the forgetting factor.

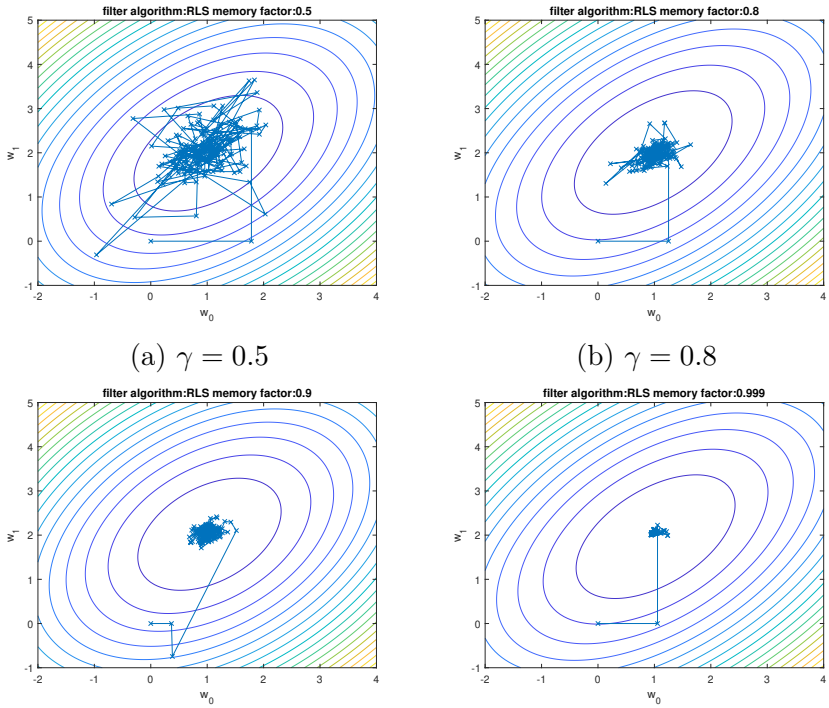
```
if strcmpi(filter_type,'RLS')
    gamma = filter.adaptation_constant;
    g = (Rx_inv_est_old*x)/(gamma^2+x'*Rx_inv_est_old*x);
    filter.Rx_inv_est=gamma^-2*(Rx_inv_est_old-g*x'*Rx_inv_est_old);
    filter.rex_est=gamma^2*rex_est_old+x*e;
    filter.w=filter.Rx_inv_est*filter.rex_est;
end
```

It can be seen in Figure 5 that for larger forgetting factors γ, i.e. a larger memory, the convergence is faster and more accurate than for small forgetting factors. This shows that a larger memory improves the estimation of $\mathbf{R}_x^{-1}$ and thus improves the weight estimation.

e

Below is the code that updates the FDAF filter on every time step. `P_est` is the estimated power matrix initialized as a square matrix filled with ones with the same dimensions as the filter size. `F` is the Fourier transformation matrix and `F_inv` is its inverse.

```
if strcmpi(filter_type,'FDAF')
    alpha=filter.adaptation_constant;
    X = filter.F*x;
    beta = 0.5;
```



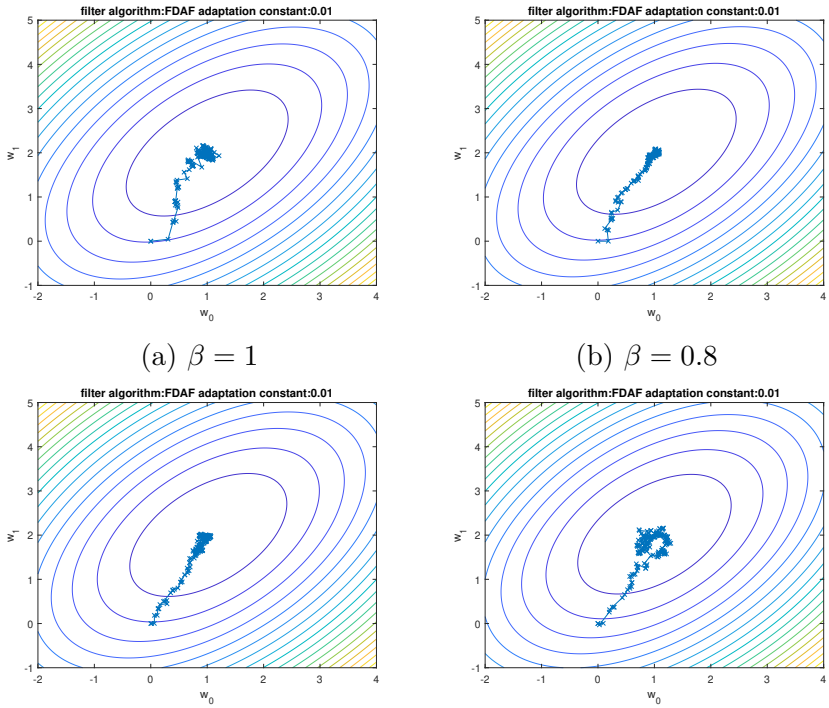
(a) $\gamma = 0.5$ (b) $\gamma = 0.8$

(c) $\gamma = 0.9$ (d) $\gamma = 0.999$

Figure 5: Recursive least squares with different forgetting factors γ.

```
    filter.P_est=beta*filter.P_est+(1-beta)*X*conj(X)'/length(x);
    W_old = filter.F_inv*w_old;
    W_new = W_old+2*alpha*1./diag(filter.P_est).*conj(X)*r;
    filter.w=filter.F*W_new;
end
```

It can be seen from figure 6 that for $\beta = 1$, which means no power normalization, that the step sizes vary a lot more, than when power normalization is used. These inconsistent step sizes result in bad convergence. For too small β such as $\beta = 0.1$, the step sizes vary a lot again, as the power normalization changes too fast per update and is inaccurate as a result. Thus, it has quite noisy converge near the optimum, which is not very nice, just like when no power normalization is used. For $\beta = 0.8$ the convergence near the optimum seems the best, but for $\beta = 0.5$, the step sizes are more consistent. Both are suitable choices depending on the desired behavior. For the FDAF algorithm it turns out that the adaptation constant α needs to be very small to have stable convergence. As a result it takes quite a while for the weights to converge, compared to RLS. However, the convergence is at least in a relatively straight line compared to NLMS.



(a) $\beta = 1$ (b) $\beta = 0.8$

(c) $\beta = 0.5$ (d) $\beta = 0.1$

Figure 6: FDAF with adaptation constant $\alpha = 0.01$ and different adaptation ratios β for the power normalization.

f

In terms of computational complexity, the simplest algorithm is LMS, as the weight update consists of only a vector multiplication, scaling and a matrix addition. The NLMS is the second simplest, as it does exactly the same, but simply adds a normalization term. FDAF and RLS are computationally the most complex algorithms due to the more sophisticated way of estimating the weights. FDAF is the same as NLMS, but converts all operations to frequency domain, making it computationally more complex. RLS has the most computational complexity of all of the discussed algorithms, due to the many matrix multiplications.

As for the accuracy, LMS is the least accurate as it has no power normalization and also does not estimate signal statistics, but just uses instantaneous estimates of the gradient. Thus the convergence path is not optimal (not a straight line) and the convergence behavior also depends on input power. NLMS slightly improves on this by normalizing the update steps by the power, such that the update steps are only dependent on the instantaneous gradient, which still is not very accurate. FDAF and RLS are both very accurate as they both converge to the Wiener solution. FDAF decorrelates the weights by converting them to frequency domain and RLS decorrelates by approximating the inverse of the autocorrelation matrix of the input signal. This decorrelation means faster convergence as the weights converge independently of the signal statistics. From the figures 5 it can be seen that RLS has faster convergence when the filters are still far away from the optimum, which is very good if the filter needs to be very quick to adapt to inputs/environments that vary over time.

To conclude, more sophisticated and complex algorithms lead to more accurate solutions in general. What algorithm is suitable depends on the accuracy requirements, available computation power, type of input signal and more. Thus we conclude that making an overall ranking on which algorithm is best is not suitable. The rankings on computational complexity and accuracy, that align with the descriptions in the above paragraphs, are tabulated in table 1.

Table 1: Comparison of complexity and accuracy of LMS, NLMS, RLS, and FDAF

	Computational Complexity	Accuracy
LMS	#1	#4
NLMS	#2	#3
RLS	#4	#1
FDAF	#3	#2