GHILENE RAYANE
LASSO NICOLAS

Tp. 3 - Advanced Signal Methods
Adaptive Noise Subtraction

1. Part I

1.1. Recalling the equations of the RLS algorithm.

The algorithm allows us to recursively find the coefficients of a given filter by minimizing a linear cost function. As in the previous practice, we are interested in reducing the noise of a given signal.

$$\mathbf{P}_n = \lambda^{-1}\left(\mathbf{P}_{n-1} - \frac{\mathbf{P}_{n-1}\mathbf{x}_n\mathbf{x}_n^T\mathbf{P}_{n-1}}{1+\lambda^{-1}\mathbf{x}_n^T\mathbf{P}_{n-1}\mathbf{x}_n}\right)$$
$$\mathbf{w}_n = \mathbf{w}_{n-1} + \mathbf{P}_n\mathbf{x}_n(e_n - \mathbf{x}_n^T\mathbf{w}_{n-1})$$

Figure 1. RLS algorithm equations

In Figure 1 we can see the two main equations of this algorithm, w being the parameter to be estimated, P the inverse correlation matrix, x the input signal and e the prediction error, all at a certain instant n. Additionally, for this algorithm we have a forgetting factor that helps us give exponentially less weight to older error samples.

1.2. RLS algorithm Implementation.

As we mentioned earlier, this is a recursive algorithm. The implementation is quite simple. taking into account that we already have the equations. These were implemented as seen in Figure 2. With initialization to 0 of all variables, except for the matrix P (which is initialized as the identity matrix), the output parameters are the signal without the added noise, the filter after the last iteration w, and the method error after applying the algorithm.

```
for n = P:N
    xn = x(n:-1:n-P+1);
    y(n) = w' * xn;
    e(n) = d(n) - y(n);
    P_n = P_n / lambda;
    P_n = P_n - (P_n * (xn * xn') * P_n) / (lambda +  xn' * P_n * xn);
    w = w + P_n * xn * e(n);
    allw(n, :) = w; % We define this for checking the filter evolution
end
```

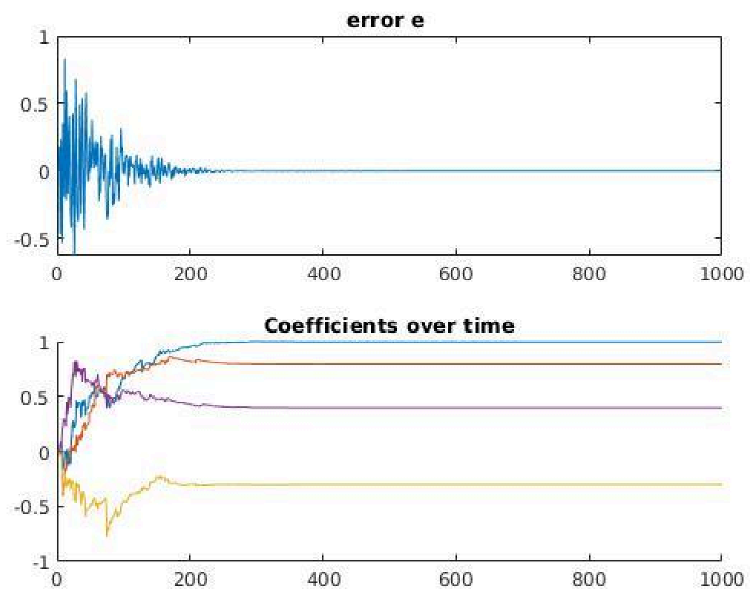
Figure 2. RLS Equations implementation

1.3. Implementation testing and validation

To test the RLS algorithm we create a white noise signal x and a signal d, filtering this with a certain finite response filter h. Our objective is to determine said filter and when testing the algorithm we obtained an almost perfect convergence, with a very good reduction in the error we observed.

```
%% Partie I

n = 1000;
x = rand(n,1) - 1/2;
b = 1;
a = [1, 0.8 , -0.3, 0.4].';
d = filter(a, b, x);
P = 4;
```

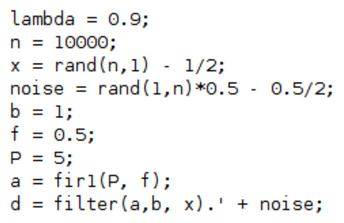
Figure 3. RLS Testing Setup



Plot 1. Error and filter coefficients over time.

1.4. Testing RLS on a simulated signal

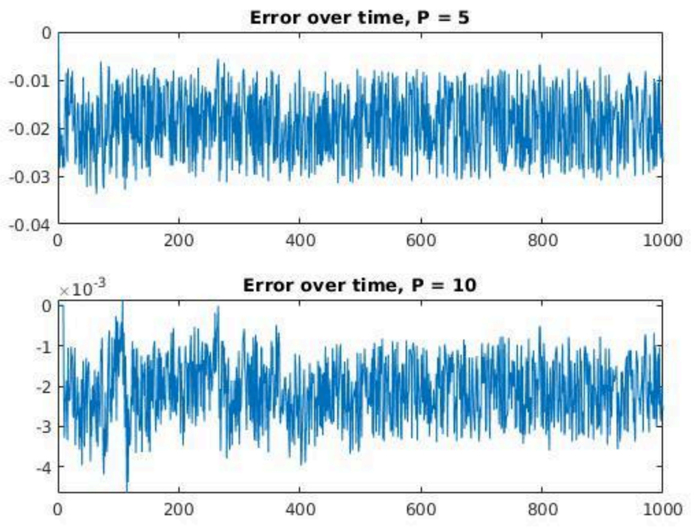
Now we want to test the algorithm on a simulated signal, so we take a white noise as input signal x, obtaining our signal d by filtering x and adding another noise. The complete setup is seen in Figure 4.



```
lambda = 0.9;
n = 10000;
x = rand(n,1) - 1/2;
noise = rand(1,n)*0.5 - 0.5/2;
b = 1;
f = 0.5;
P = 5;
a = fir1(P, f);
d = filter(a,b, x).' + noise;
```

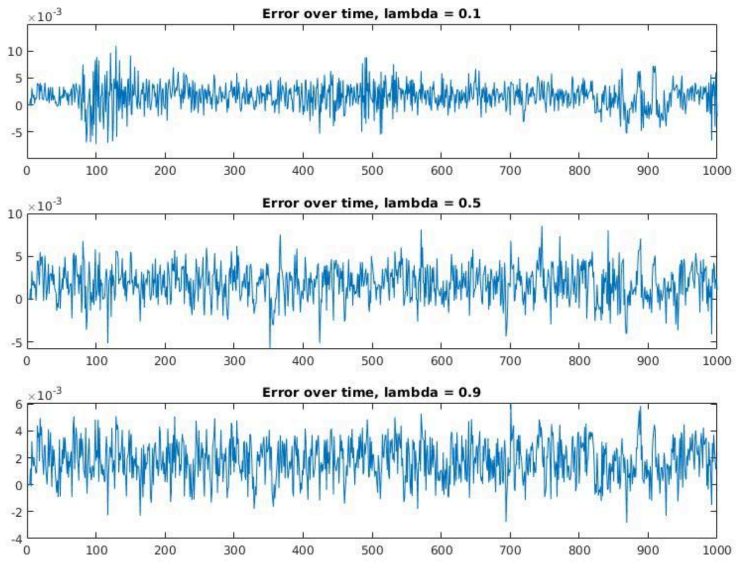
Figure 4. Error and filter coefficients over time.

In the axes of the different errors calculated for two values of P it is observed how a greater error is obtained by increasing the order of the filter, because as in any estimation problem, the greater the complexity, the greater the probability of having errors due to aggregation. of smaller errors.



Plot 2. Error and filter coefficients over time.

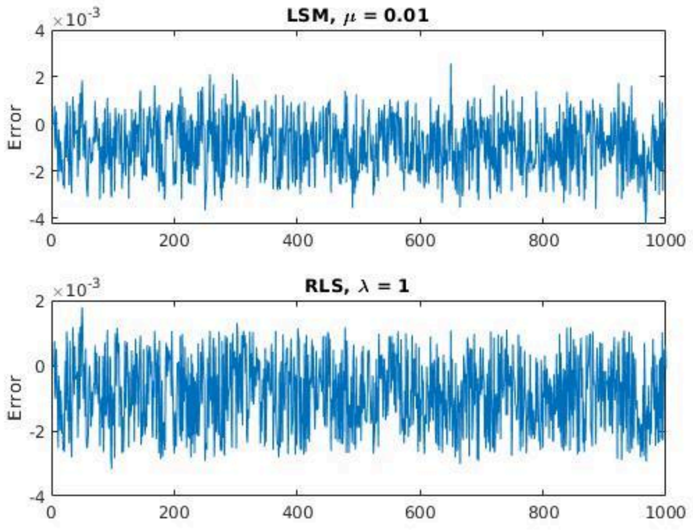
In the Plot 2, we can see how the error behaves over time. If we pay attention to the axes, we see how this error decreases as lambda increases, being at a minimum with lambda = 1, this being the special case of growing window RLS algorithm.



Plot 3. Error and filter coefficients over time.

1.5. Comparison of LMS and RLS performances.

When comparing the different algorithms, we want to select the parameter that we consider optimal for each of them, in this case Learning rate = 0.01, for LMS and Lambda = 1 for RLS. By observing the values that the error takes over time (Plot 4) it can be seen that the RLS error is lower than that of LMS, something to be expected considering the complexity that the first method can take.



```
VarErrorLMS =

   2.0843e-06

VarErrorRLS =

   1.2679e-06
```

Plot 4. Error for LMS and RLS over time.

2. Part II

2.1. Adaptive Noise Canceller Implementation with two input signals

In this part, we will implement the following adaptive noise canceller

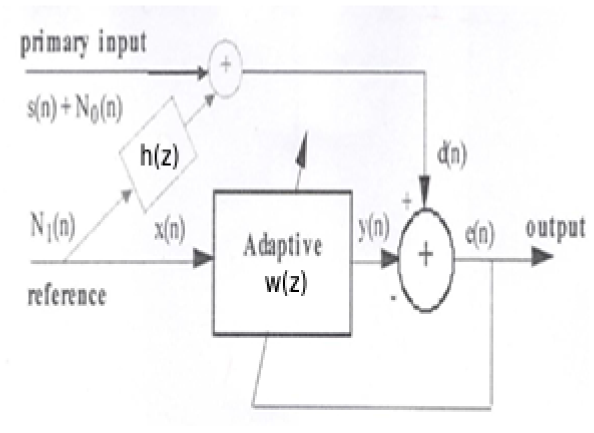
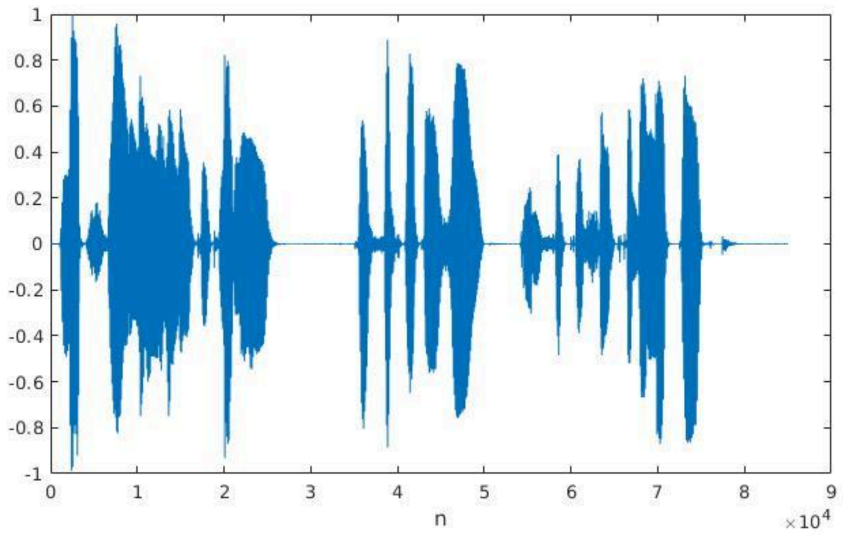
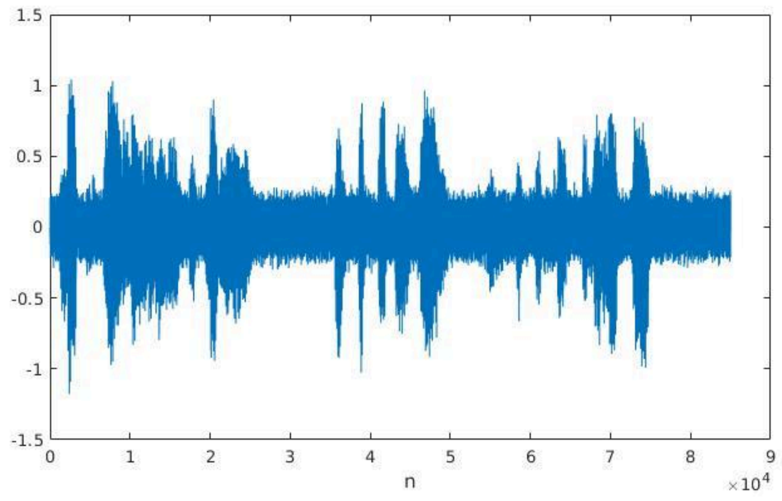


Figure 5. Adaptive Noise Canceller

s(n): Input signal
N0(n): Noise
N1(n): Reference Signal
N0(n): Filtered N1(n) using a finite response filter

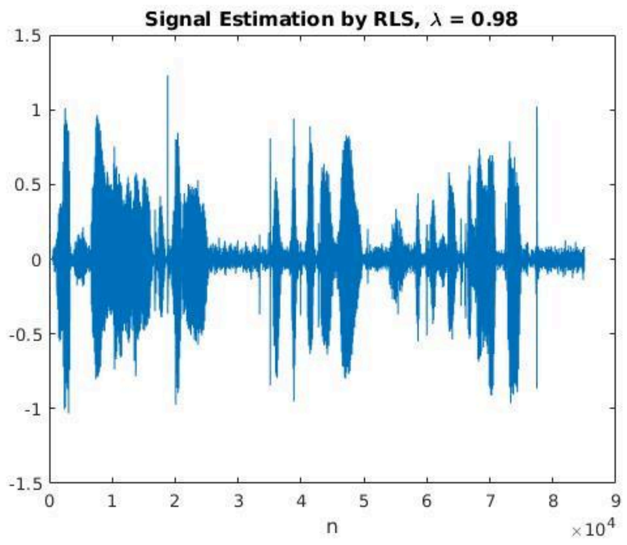


Plot 5. Signal S(n)



Plot 6. Signal S(n) + N0

After filtering we obtained the following estimated signal:



Plot 7. Signal estimated by RLS

Using a Lambda so close to 1 we get a high-fidelity estimation although this method is memory intensive, in our case the sampling frequency is low enough that we can process it without running into issues.

If we had an audio file with a large sampling frequency we wouldn't have been able to process it, that's why it's recommended to choose a lambda in [0.98, 1]

3. Part III

3.1. The NLMS Algorithm

Now we will repeat the process again, but this time with a new algorithm, or rather, a revision of an algorithm that we had already worked with in the previous session, the NLMS algorithm.

When observing the equations determined with this algorithm (Figure 6) we see that the only thing that changes is that we now normalize our input vector in each iteration, adding a very low epsilon value to avoid dividing by zero.

$$y(n) = \mathbf{w}^T(n)\mathbf{x}(n)$$
$$e(n) = d(n) - y(n)$$
$$\mathbf{w}(n+1) = \mathbf{w}(n) + \mu \cdot \frac{\mathbf{x}(n)e(n)}{\|\mathbf{x}(n)\|^2 + \epsilon}$$

Figure 6. NLMS algorithm equations

3.2. NLMS Implementation

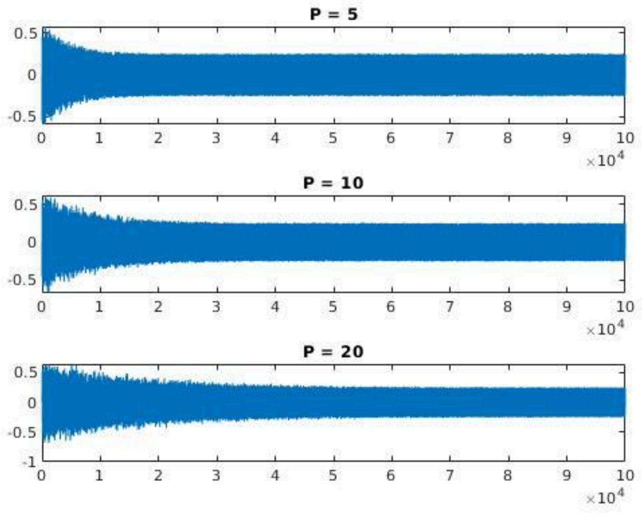
We take the equations described above and implement them in MatLab. As in the other instances, all vectors are initialized to 0, and the outputs are the vector of filter coefficients, w; the error vector, e; and finally the exit, and.

```
w = zeros(P, 1);
y = zeros(N, 1);
e = zeros(N, 1);
allw = zeros(N, P);

P_n = eye(P);

for n = P:N
    xn = x(n:-1:n-P+1);
    y(n) = w' * xn;
    e(n) = d(n) - y(n);
    P_n = P_n / lambda;
    P_n = P_n - (P_n * (xn * xn') * P_n) / (lambda +  xn' * P_n * xn);
    w = w + P_n * xn * e(n);
    allw(n, :) = w; % We define this for checking the filter evolution
end
```

Figure 7. NLMS algorithm MatLab Implementation

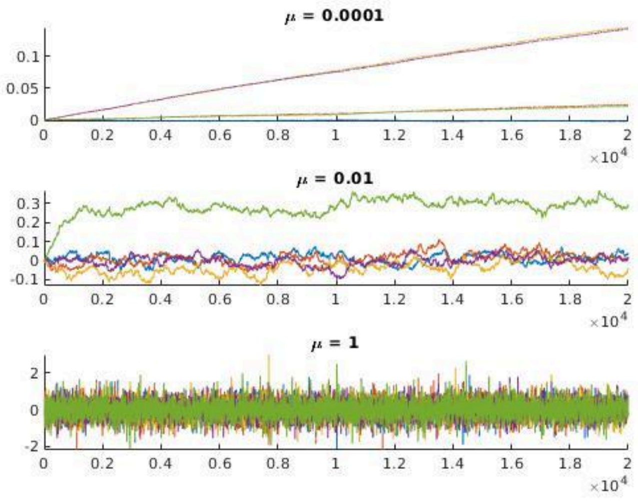
3.3. Testing the NLMS Implementation

In order to test our implementation, we will apply the algorithm to the same signal. with the same noise but varying the filter coefficients. As a measure we have the error e and the results can be observed in the Plot 8. We see how the error behaves when increasing the filter order and we conclude that convergence may take longer at higher filter values, however, convergence is always reached.



Plot 8. NLMS Error with different P values

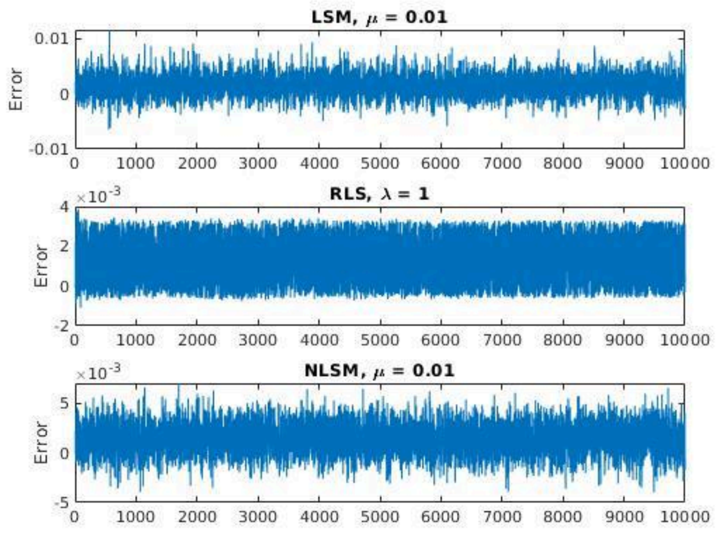
In the case of the convergence speed, we graph the values of the filter coefficients, with a fixed P, but with different values for the learning rate. As expected, a very low value does not fully converge to the expected values, a value that is too high only moves around the real value without ever converging, but a value close to the optimal allows us to gradually get closer to the objective of the estimate.



Plot 9. NLMS Convergence with different learning rates

3.4. Comparison of performances of NLMS, LMS, and RLS

To conclude, we have the comparison between the error of the three methods, each with a parameter value considered optimal, and trying to try a filter of fixed order P for the three different methods. The results, once again, are to be expected. The algorithm with the best performance is RLS, with a lower error variance than the other methods. Then follows NLMS and finally LMS. The more complex one of the algorithms is, the more memory and resources it occupies; however, its performance will be better, although it cannot be fully appreciated in this practice due to the simplicity of the experiment.



```
VarErrorLMS =

   3.8691e-06

VarErrorNLMS =

   2.4152e-06

VarErrorRLS =

   1.2862e-06
```

Plot 10. Different methods comparison.

End.