Accuracy-Constrained Computational Tradeoffs of LMS, NLMS, and RLS under Correlated Inputs

Praneel Patel
Oak Park High School
Oak Park, California, USA
praneelpatel0@gmail.com

November 2, 2026

Abstract

Adaptive filters usually get compared in one of two ways: by how many iterations they need to learn, or by how much work a single iteration takes. An engineer needs both numbers at once, because what matters is the total computation required to reach a given accuracy. I measured that for three common algorithms: least-mean-squares (LMS), normalized LMS (NLMS), and recursive least squares (RLS). Each one was tuned to reach a shared accuracy target as fast as it could, using separate tuning and test trials. I then measured the computation needed to reach the target for input correlations ρ from 0 to 0.99 and filter lengths L from 8 to 128, with bootstrap confidence intervals. At the main target, RLS had the lowest cost in 8 of 20 conditions, LMS in 8, and NLMS in 0; 4 conditions were statistical ties. White and mildly correlated inputs mostly favored LMS. Once $\rho \geq 0.9$, LMS and NLMS needed far more iterations or never reached the target, and RLS became the cheapest choice at nearly all tested lengths even though each of its updates costs more. The result is a map of which algorithm is cheapest for a given input and filter length, along with a list of where that map stops being trustworthy.

1 Introduction

Adaptive filters change their own coefficients as new data comes in. People use them for noise cancellation, echo cancellation, and communication systems [5, 9]. This paper looks at three standard ones: LMS, NLMS, and RLS.

LMS is popular because its update is simple and cheap [8]. But it learns very slowly when the input signal is strongly correlated. NLMS divides the LMS update by the input energy, which makes the step size easier to pick [7]. RLS learns much faster and hardly notices input correlation, though one update costs roughly L^2 operations where LMS costs roughly L [1, 5, 6].

Engineers already know all this, and it leaves them with a real decision. RLS needs less iterations but each one costs more. So which algorithm uses less total computation to reach a required accuracy? I expected the answer to depend on two things: the filter length, which sets what an RLS update costs, and the input correlation, which sets how slow LMS gets.

Prior work. Textbooks cover the convergence speed, steady-state error, and operation count of each algorithm [1, 5, 6], but they mostly treat these as separate topics. Eleftheriou and Falconer studied the tracking and steady-state behavior of RLS and compared its speed-versus-accuracy tradeoff with LMS

[3]. Eweda compared RLS, LMS, and the sign algorithm for tracking randomly time-varying channels [4]. Newer comparisons keep appearing in specific application areas such as active noise control and wireless channel equalization. As far as I can tell they agree that LMS and RLS settle at similar steady-state error, while RLS needs far fewer samples to converge when there is intersymbol interference or highly correlated noise. Most of them plot learning curves against iterations at step sizes picked by hand, though, and never bound the computational cost. Table 1 summarizes how each study handles the comparison. What this paper adds is a map of where the total cost crosses over, measured at one fixed accuracy target that all three algorithms share.

Table 1: Comparison of adaptive filter benchmark methodologies.

Study	Algorithms	Input Correlation	Primary Cost Metric
Eleftheriou (1986)	LMS, RLS	None (White)	Tracking MSE
Eweda (1994)	LMS, RLS, Sign	None (White)	Convergence rate
Recent applied benchmarks	LMS, NLMS, RLS	Fixed scenarios	Iterations to converge
This Work	**LMS, NLMS, RLS**	**Swept ($\rho = 0$ to 0.99)**	**Total computation (C_{target})**

Research question. Under what input correlations and filter lengths does RLS reduce the computation needed to reach a shared accuracy target, compared with carefully tuned LMS and NLMS?

Hypothesis. Before running the main experiment I expected RLS to win for short filters and strongly correlated inputs, to lose on long filters because of its L^2 cost, and to have that crossover move toward longer filters as the correlation went up. That was only a guess, and I set the experiments up so they could prove it wrong.

Contributions. This paper:

1. compares the three algorithms at a *shared* accuracy target, not at each algorithm's own final error;
2. tunes each algorithm for that target on development trials, then reports results from separate test trials;
3. measures the computation needed to reach the target across $\rho \in \{0, 0.5, 0.9, 0.99\}$ and $L \in \{8, \ldots, 128\}$, with bootstrap confidence intervals, a second target, and an operation-count version of the same comparison that does not depend on hardware; and
4. releases code that recreates every figure, table, and number in the paper.

The study only covers simulated data, a static unknown system, Gaussian inputs, and offline processing in MATLAB. Section 5 goes through these limits.

2 Background

2.1 Signal Model

Let $x(n)$ be the input signal. For a filter with L coefficients, the input vector is

$$\mathbf{u}(n) = [x(n), x(n-1), \ldots, x(n-L+1)]^T. \tag{1}$$

The input is a stationary, unit-variance AR(1) process:

$$x(n) = \rho\, x(n-1) + \sqrt{1-\rho^2}\,\xi(n), \tag{2}$$

where $\xi(n)$ is white Gaussian noise with unit variance. The value ρ sets how strongly neighboring samples are correlated, and $\rho = 0$ gives white input. The first sample comes from the stationary distribution, so no burn-in is needed. The input correlation matrix is $\mathbf{R} = \mathbb{E}[\mathbf{u}(n)\mathbf{u}^T(n)]$, with entries $R_{ij} = \rho^{|i-j|}$.

The desired signal is

$$d(n) = \mathbf{w}_o^T \mathbf{u}(n) + v(n), \tag{3}$$

where $\mathbf{w}_o$ is the unknown system and $v(n)$ is white Gaussian measurement noise with variance σ_v^2.

Signal-to-noise ratio. The power of the clean output $\mathbf{w}_o^T\mathbf{u}(n)$ is $\mathbf{w}_o^T\mathbf{R}\mathbf{w}_o$. That equals 1 for white input and a unit-norm $\mathbf{w}_o$, but not for correlated input. So the SNR is defined as

$$\text{SNR} = \frac{\mathbf{w}_o^T\mathbf{R}\mathbf{w}_o}{\sigma_v^2}, \tag{4}$$

and in every trial σ_v^2 is set to make the SNR exactly 20 dB.

The error the adaptive filter uses is the a priori error, measured before the coefficients are updated:

$$e(n) = d(n) - \mathbf{w}^T(n)\mathbf{u}(n). \tag{5}$$

2.2 The Three Algorithms

LMS. The update is

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \mu\, e(n)\mathbf{u}(n). \tag{6}$$

A larger step size μ learns faster but leaves more error afterward, and a step that is too large makes the filter unstable.

NLMS. The update divides by the current input energy:

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \frac{\mu\, e(n)\mathbf{u}(n)}{\varepsilon + \|\mathbf{u}(n)\|^2}. \tag{7}$$

RLS. The gain vector, coefficient update, and inverse-correlation update are

$$\mathbf{k}(n) = \frac{\mathbf{P}(n-1)\mathbf{u}(n)}{\lambda + \mathbf{u}^T(n)\mathbf{P}(n-1)\mathbf{u}(n)}, \tag{8}$$

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \mathbf{k}(n)e(n), \tag{9}$$

$$\mathbf{P}(n) = \lambda^{-1}\left[\mathbf{P}(n-1) - \mathbf{k}(n)\mathbf{u}^T(n)\mathbf{P}(n-1)\right], \tag{10}$$

with $\mathbf{P}(0) = \delta^{-1}\mathbf{I}$. The forgetting factor $\lambda \le 1$ sets how quickly old data is forgotten. After each update, $\mathbf{P}$ is replaced by $(\mathbf{P}+\mathbf{P}^T)/2$ so it stays exactly symmetric, which stops rounding errors from piling up.

Cost per update. Counting multiplications and divisions in my implementations, one update costs $2L+1$ for LMS, $3L+2$ for NLMS, and $4L^2+4L$ for RLS (the symmetrizing step adds L^2). Operation counts are not run time, though. In MATLAB the fixed overhead of each loop iteration dominates for small filters. I report both.

2.3 Why Correlation Slows LMS

LMS learns along each eigenvector of $\mathbf{R}$ at a speed set by μ times that eigenvalue [5]. The largest eigenvalue limits the step size, and the smallest one decides how long learning takes. For the AR(1) input in Eq. (2), the ratio of the largest to the smallest eigenvalue approaches $((1+\rho)/(1-\rho))^2$ as L grows. That is about 9 at $\rho = 0.5$, 361 at $\rho = 0.9$, and nearly 40,000 at $\rho = 0.99$. RLS uses $\mathbf{P}(n) \approx \mathbf{R}^{-1}$ to cancel this spread, which is why its learning speed hardly changes with ρ.

2.4 Performance Measures

The mean-square error (MSE) is $J(n) = \mathbb{E}[e^2(n)]$. The noise power changes from trial to trial, so the main measure is the *normalized* MSE $J(n)/\sigma_v^2$. Its lowest possible value is 1, reached when the filter matches the system exactly and only measurement noise is left.

The mean-square deviation (MSD) is

$$D(n) = \mathbb{E}\left[\|\mathbf{w}(n) - \mathbf{w}_o\|^2\right]. \tag{11}$$

MSE says how well the filter *predicts* $d(n)$, while MSD says how close the *coefficients* are to the true system. With correlated input, two filters can have similar MSE and very different MSD, since coefficient errors in directions with little input energy hardly touch the output. So MSD is a supporting measure here, not a replacement for MSE.

2.5 LMS Steady-State Theory

For white Gaussian input with power σ_x^2, the standard independence-theory result for the steady-state MSE of LMS is [5, 6]

$$J_\infty = \frac{\sigma_v^2}{1 - \dfrac{L\mu\sigma_x^2}{2 - \mu\sigma_x^2}}. \tag{12}$$

This assumes the input vector is independent of the current coefficient error, that the input is white and Gaussian, and that μ is small enough to keep the denominator positive (here, $\mu < 2/(L+1)$ when $\sigma_x^2 = 1$). Independence is not exactly true for a tapped delay line, so the formula is an approximation and should work best for small μ. In this paper it is only a check that the simulation code is right.

3 Method

3.1 A Shared Accuracy Target

Comparing algorithms by how fast they reach their *own* final error is not fair when the final errors differ, since a filter that settles higher gets an easier finish line. So all three algorithms share one target:

$$\frac{J(n)}{\sigma_v^2} \le T, \qquad T = 1.10. \tag{13}$$

Put simply, the filter has to get within 10% of the best possible MSE. I also use a second target, $T = 1.25$, to check that the conclusions do not hinge on one choice. Both targets were fixed before the final test runs.

The learning curve is the average of $e^2(n)/\sigma_v^2$ over all test trials, then averaged again over blocks of 20 iterations. The *iterations to target*, N_{target}, is the end of the first block after which the curve stays at or

below T for at least 200 iterations in a row. If that never happens inside the 20000-iteration horizon, the result is recorded as "not reached" (n.r.) rather than as missing data.

3.2 Computation to Target

The computation needed to reach the target is

$$C_{\text{target}} \approx N_{\text{target}} \times (\text{measured time per update}). \tag{14}$$

I checked this approximation by timing runs of exactly N_{target} iterations directly, initialization included. Across all conditions the directly timed cost was within 58% of C_{target} at most. As a hardware-independent version of the same comparison, C_{target} is also computed with operation counts in place of measured time.

C_{target} measures *computation*, like processing recorded data offline. It is not real-time delay. In a live system sampled at rate f_s, collecting N_{target} samples takes N_{target}/f_s seconds whichever algorithm runs, so the real question is whether each update finishes before the next sample shows up. I only study the offline case.

3.3 Tuning Without Overfitting

Each algorithm has one main parameter, and the three are not directly comparable, so I tuned each separately. The grids scale with L so they treat every filter length the same way:

- LMS: $\mu = c/L$ with $c \in \{0.01, 0.02, 0.05, 0.1, 0.2, 0.3, 0.5\}$;
- NLMS: $\mu \in \{0.02, 0.05, 0.1, 0.2, 0.3, 0.5, 0.8\}$, with $\varepsilon = 10^{-6}$;
- RLS: $\lambda = 1 - c/L$ with $c \in \{0, 0.01, 0.02, 0.05, 0.1, 0.2, 0.5\}$, with $\delta = 0.01$. The value $c = 0$ gives $\lambda = 1$, the natural choice for a system that does not change.

Every grid has seven values, so the tuning effort is equal between the three algorithms. For each (ρ, L) condition I ran every grid value on 100 *development* trials and kept the one with the smallest N_{target}. If nothing reached the target, I kept the value with the lowest final error and marked it. Those settings were then frozen and evaluated on 200 new *test* trials that never touched the tuning. Everything in Section 4 comes from the test trials. Settings that landed on the edge of a grid are marked in Table 8 (Appendix A).

3.4 Trials and Random Numbers

In each trial a new unknown system $\mathbf{w}_o$ is drawn from a Gaussian distribution and scaled to unit norm, and new input and noise signals are generated. The results therefore average over many systems, not one lucky one. Inside a trial, all three algorithms see exactly the same system, input, and noise, which makes the comparison *paired*. Each part of the study has its own seed, equal to the base seed 2026 plus a fixed offset (development trials for a condition use a different offset than test trials, for example). Any single part can be rerun alone and give the same numbers.

3.5 Uncertainty

Confidence intervals come from the bootstrap [2]: the 200 test trials were resampled with replacement 1000 times. I resampled whole trials, not single time samples, because samples inside one run are correlated. Every resample used the same trial indices for all three algorithms, which keeps things paired.

Since N_{target} comes from the *average* learning curve, it was recomputed inside every resample: build the resampled curve, then find where it crosses the threshold. Resamples where the curve never met the target stayed as "not reached". (This is different from finding each run's crossing time and averaging those.) A method is the clear winner in a condition if it has the lowest C_{target} in at least 95% of the resamples where at least one method reached the target. If none does, the condition is reported as a tie among every method that won at least 5% of the time. MSE values are always averaged in linear units first and converted to dB afterward.

3.6 Timing Protocol

Time per update was measured separately from the accuracy runs. For each algorithm and filter length I generated the data first, did one warm-up run so MATLAB could compile the code, and then recorded 15 timed runs of 20000 iterations each. Data generation, error bookkeeping, and plotting stayed outside the timed region, and all arrays were preallocated. I report the median time per update and its interquartile range. All simulations ran in MATLAB 26.1.0.3312084 (R2026a) Update 4 on macOS (Apple M2, MACA64), in double precision, with 8 computational threads.

3.7 Secondary Experiments

LMS theory check. With white input and $L = 16$, LMS ran at five step sizes for 8000 iterations in each of 200 trials. I averaged the last 2000 iterations of each trial and compared that with Eq. (12).

Tracking. In the tracking experiment the unknown system drifts as a random walk,

$$\mathbf{w}_o(n+1) = \mathbf{w}_o(n) + \mathbf{q}(n), \qquad \mathbf{q}(n) \sim \mathcal{N}(\mathbf{0}, \sigma_q^2 \mathbf{I}), \tag{15}$$

with white input, $L = 16$, $\sigma_v^2 = 10^{-2}$, and five values of σ_q^2 from 10^{-9} to 10^{-5}. Here σ_q^2 is the drift *per coefficient*, so the expected total drift per step is $L\sigma_q^2$. Each algorithm was tuned on 30 development trials (LMS μ from 0.0005 to 0.05; NLMS μ from 0.01 to 1; RLS λ from 0.9 to 1). The chosen setting was then tested on 100 new trials, using the last 1000 of 4000 iterations.

Table 2: Main simulation settings.

Setting	Value
Input correlation ρ	0, 0.5, 0.9, 0.99
Filter length L	8, 16, 32, 64, 128
Output SNR	20 dB (every trial)
Main / second target T	1.10 / 1.25
Horizon	20000 iterations
Block size / required time below target	20 / 200 iterations
Development / test trials per condition	100 / 200
Bootstrap resamples	1000
RLS δ; NLMS ε	0.01; 10^{-6}
Base random seed	2026

4 Results

4.1 Computation to Reach the Target

Figure 1 and Table 3 show C_{target} for every condition at the main target. Take $\rho = 0.9$ and $L = 16$: LMS needed 6380 iterations, NLMS needed 5060, and RLS needed 240. In computation time that came to 1.2 [1.18, 1.5], 1.11 [0.928, 1.76], and 0.255 [0.191, 0.467] ms (95% confidence intervals in brackets). At $L = 128$ and the same ρ, the lowest cost was: RLS (100%).

With white input ($\rho = 0$), the $\mathcal{O}(L^2)$ cost of RLS made it more expensive than LMS at every filter length. At $\rho = 0.5$ RLS was competitive on short filters and tied with LMS at $L = 8$ and $L = 16$, but LMS was the clear winner from $L \geq 32$ on. Stronger correlation moved the advantage a long way toward RLS, because it slows the gradient-based methods so much. At $\rho = 0.9$, RLS was the clear cheapest option for $L \leq 16$.

When filter length and correlation both grow, LMS and NLMS eventually fail to reach the target inside the 20000-iteration horizon. At $\rho = 0.9$ LMS failed for $L \geq 64$ and NLMS failed at $L = 128$. At $\rho = 0.99$ LMS failed at every filter length I tested and NLMS failed for $L \geq 32$, so RLS was the cheapest method across the whole range of L.

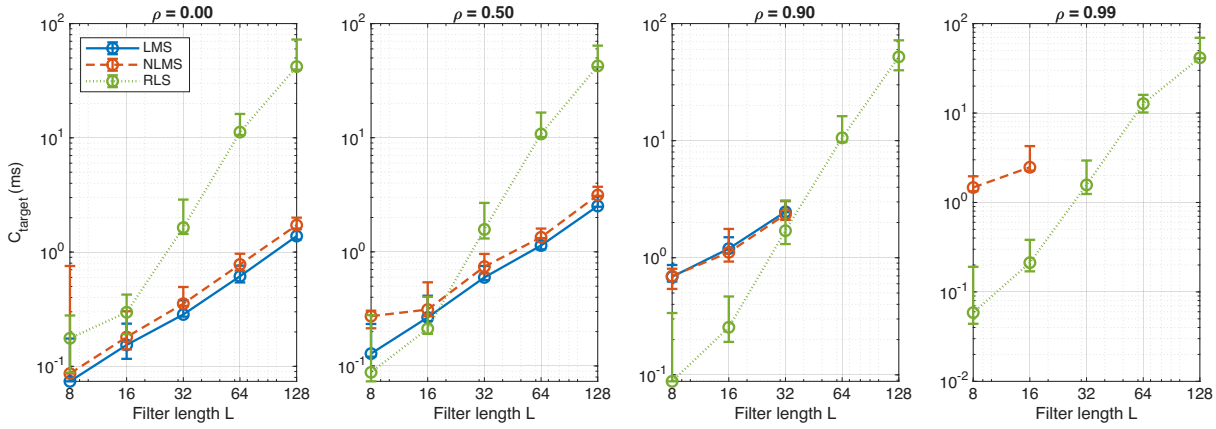

Figure 1: Computation to reach $J(n)/\sigma_v^2 \leq 1.10$ versus filter length, for each input correlation. Error bars are 95% bootstrap confidence intervals. Missing points mean the target was not reached within the horizon.

Table 3: Iterations and computation to reach the main target ($T = 1.10$) on the test trials. n.r. = not reached within 20000 iterations. "Lowest cost" gives the clear winner and the share of bootstrap resamples it won, or a tie.

		N_{target} (iterations)			C_{target} (ms) [95% CI]			
ρ	L	LMS	NLMS	RLS	LMS	NLMS	RLS	Lowest cost
0.00	8	380	380	240	0.0739 [0.0739, 0.175]	0.0865 [0.0865, 0.756]	0.176 [0.088, 0.279]	LMS (97%)
	16	820	820	280	0.154 [0.116, 0.236]	0.18 [0.141, 0.303]	0.297 [0.191, 0.425]	LMS (98%)
	32	1400	1400	500	0.284 [0.272, 0.378]	0.353 [0.338, 0.495]	1.64 [1.44, 2.88]	LMS (100%)
	64	3080	3080	1040	0.614 [0.542, 0.762]	0.779 [0.688, 0.967]	11.2 [10.6, 16.2]	LMS (100%)
	128	5780	5780	1600	1.38 [1.29, 1.59]	1.72 [1.61, 2]	42 [39.4, 72.5]	LMS (100%)
0.50	8	660	1200	120	0.128 [0.121, 0.233]	0.273 [0.214, 0.305]	0.088 [0.0733, 0.279]	tie: LMS/RLS
	16	1420	1420	200	0.266 [0.248, 0.413]	0.312 [0.273, 0.541]	0.212 [0.191, 0.403]	tie: LMS/RLS
	32	2940	2940	480	0.597 [0.565, 0.751]	0.742 [0.646, 0.959]	1.57 [1.31, 2.68]	LMS (98%)
	64	5700	5300	1000	1.14 [1.05, 1.32]	1.34 [1.32, 1.6]	10.8 [10.1, 16.6]	LMS (100%)
	128	10560	10560	1620	2.52 [2.48, 3.05]	3.14 [3.1, 3.72]	42.6 [41.5, 64.1]	LMS (100%)
0.90	8	3540	3060	120	0.688 [0.626, 0.867]	0.696 [0.542, 0.806]	0.088 [0.088, 0.337]	RLS (100%)
	16	6380	5060	240	1.2 [1.18, 1.5]	1.11 [0.928, 1.76]	0.255 [0.191, 0.467]	RLS (100%)
	32	12160	9300	520	2.47 [2.45, 3.05]	2.35 [2.1, 3.05]	1.7 [1.31, 3.08]	tie: NLMS/RLS
	64	n.r.	n.r.	980	n.r.	n.r.	10.6 [9.71, 16.2]	tie: NLMS/RLS
	128	n.r.	n.r.	1980	n.r.	n.r.	52 [39.9, 72]	RLS (100%)
0.99	8	n.r.	6460	80	n.r.	1.47 [1.34, 1.96]	0.0587 [0.044, 0.191]	RLS (100%)
	16	n.r.	11260	200	n.r.	2.48 [2.27, 4.27]	0.212 [0.17, 0.382]	RLS (100%)
	32	n.r.	n.r.	480	n.r.	n.r.	1.57 [1.24, 2.95]	RLS (100%)
	64	n.r.	n.r.	1180	n.r.	n.r.	12.7 [10.1, 16]	RLS (100%)
	128	n.r.	n.r.	1580	n.r.	n.r.	41.5 [41, 69.3]	RLS (100%)

4.2 Regime Map and Robustness

Figure 2 shows which method had the lowest cost in each condition. At the main target, RLS was the clear winner in 8 of 20 conditions, LMS in 8, and NLMS in 0. There were 4 ties, and in 0 conditions no method reached the target.

Table 4 checks the map two ways. With the looser target $T = 1.25$ (re-tuned for it), the label stayed the same in 17 of 20 conditions. With operation counts in place of measured time, it stayed the same in 16 of 20 conditions.

Switching from measured time to operation counts changed the lowest-cost method in 4 of the 20 conditions. RLS lost ground at the shorter filter lengths, going from 8 clear wins in measured time to only 6 with pure operation counts, while LMS went from 8 wins to 10. The reason is MATLAB's fixed overhead per iteration. At small L that overhead dominates the measured time and hides how steeply the $\mathcal{O}(L^2)$ cost of RLS grows. Judged on operation counts alone, the point where RLS gets too expensive moves toward shorter filters and higher correlations.

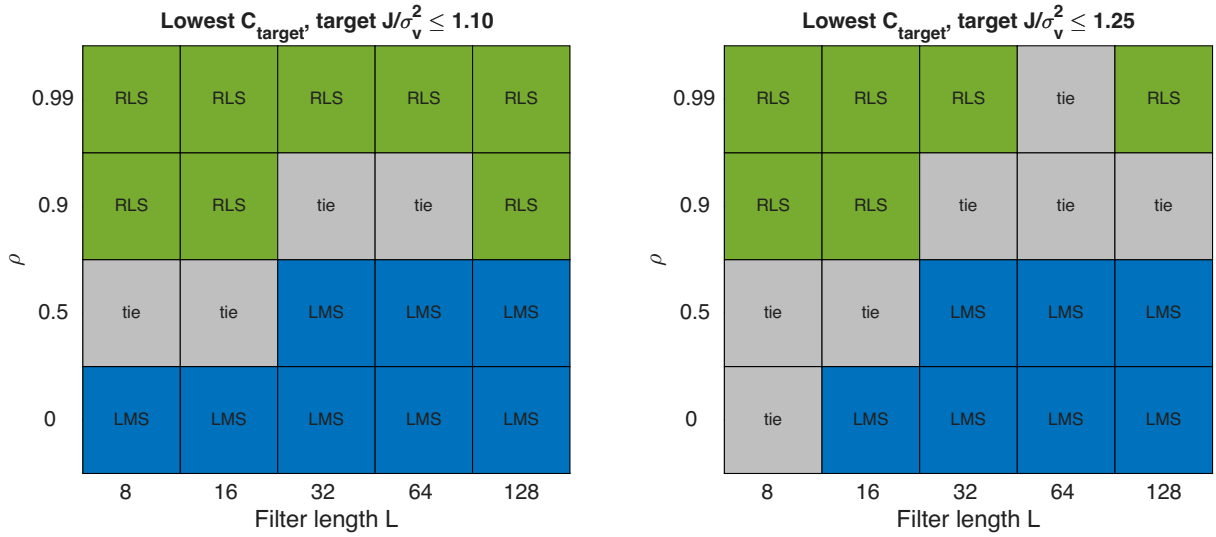

Figure 2: Method with the lowest measured computation to target in each condition, for the main target (left) and the second target (right). "tie" means no method won in at least 95% of bootstrap resamples.

Table 4: Robustness of the lowest-cost method to the choice of target and to the cost measure.

		Measured time		Operation count
ρ	L	Target 1.10	Target 1.25	Target 1.10
0.00	8	LMS (97%)	tie: LMS/RLS	LMS (100%)
	16	LMS (98%)	LMS (99%)	LMS (100%)
	32	LMS (100%)	LMS (99%)	LMS (100%)
	64	LMS (100%)	LMS (100%)	LMS (100%)
	128	LMS (100%)	LMS (98%)	LMS (100%)
0.50	8	tie: LMS/RLS	tie: RLS	LMS (100%)
	16	tie: LMS/RLS	tie: LMS	LMS (99%)
	32	LMS (98%)	LMS (98%)	LMS (100%)
	64	LMS (100%)	LMS (99%)	LMS (100%)
	128	LMS (100%)	LMS (100%)	LMS (100%)
0.90	8	RLS (100%)	RLS (100%)	tie: LMS/RLS
	16	RLS (100%)	RLS (100%)	tie: LMS/NLMS/RLS
	32	tie: NLMS/RLS	tie: NLMS/RLS	tie: LMS/NLMS
	64	tie: NLMS/RLS	tie: LMS/NLMS	tie: NLMS/RLS
	128	RLS (100%)	tie: LMS/NLMS	RLS (100%)
0.99	8	RLS (100%)	RLS (100%)	RLS (100%)
	16	RLS (100%)	RLS (100%)	RLS (100%)
	32	RLS (100%)	RLS (100%)	RLS (100%)
	64	RLS (100%)	tie: NLMS/RLS	RLS (100%)
	128	RLS (100%)	RLS (100%)	RLS (100%)

4.3 Learning Curves and Coefficient Error

Figure 3 plots the same learning curves against computation time instead of iteration number, so the tradeoff is visible directly. Figure 4 shows them against iteration number with 95% bootstrap bands, along with the matching MSD curves.

At $\rho = 0.9$ and $L = 16$, the final MSD was -33.0 dB for LMS, -28.1 dB for NLMS, and -41.4 dB for RLS, even though all three were tuned to reach the same MSE target.

In the learning curves RLS drops almost vertically, while LMS and NLMS decay slowly and gradually. The MSD ranking is also different from the MSE ranking. Very different, actually. All three reached the same 1.10 MSE target, but RLS ended with a much lower (better) coefficient error. The reason is the eigenvalue spread of $\mathbf{R}$ at $\rho = 0.9$: coefficient errors along eigenvectors with near-zero eigenvalues barely change the output MSE, so LMS and NLMS leave large errors sitting in those low-energy directions. RLS tracks the inverse correlation matrix and undoes the spread, which lets it get closer to the true system for the same MSE.

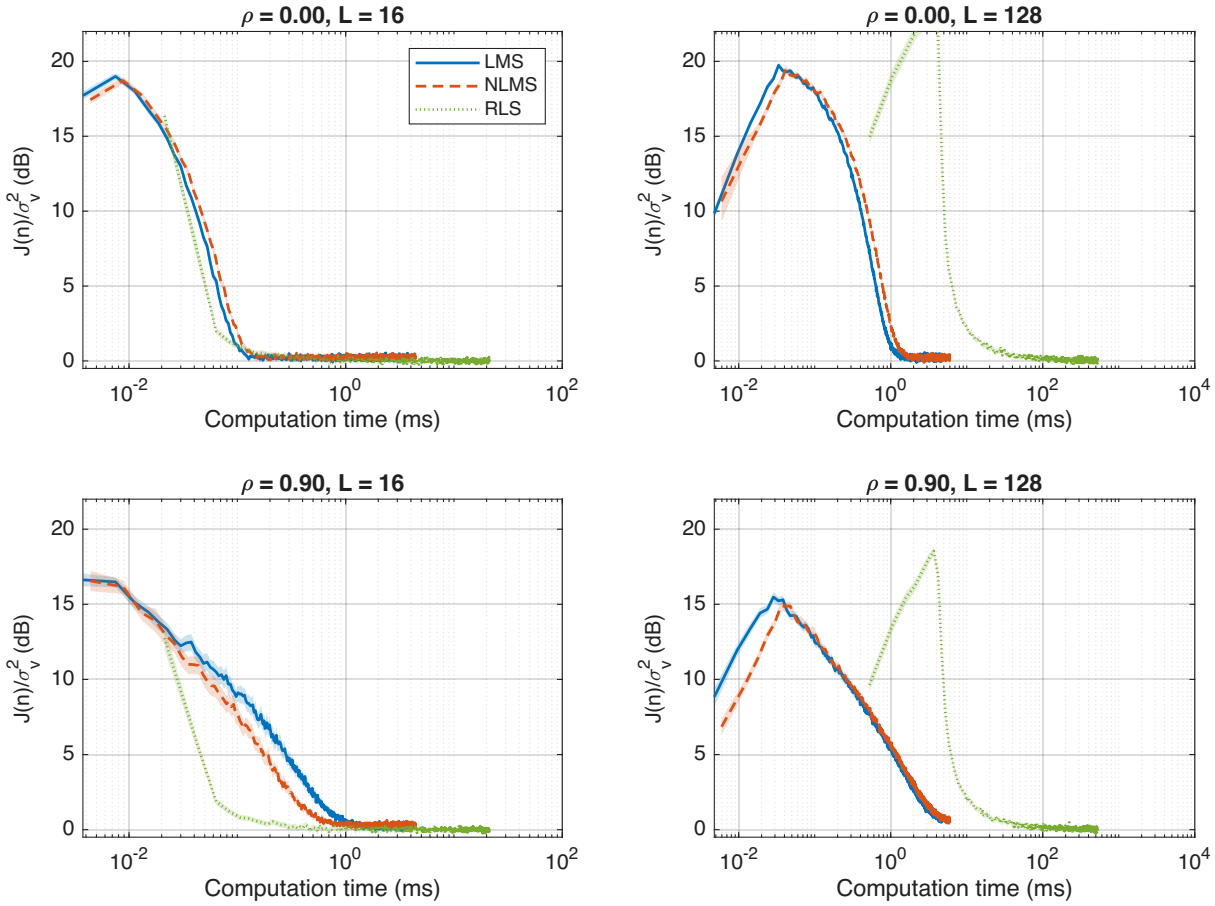

Figure 3: Normalized MSE versus computation time for tuned filters at representative conditions. Shaded bands are 95% bootstrap intervals. The dash-dot line is the target.

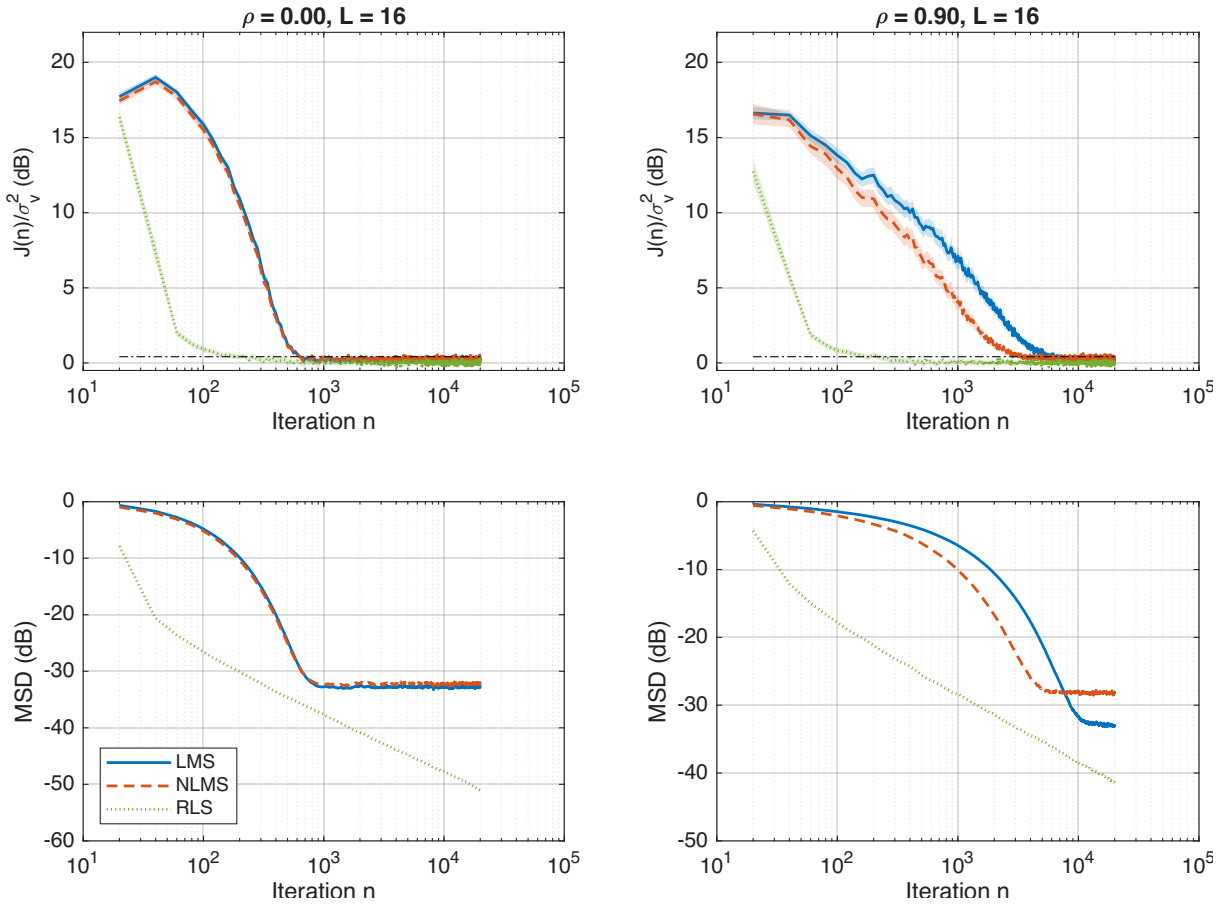

Figure 4: Top: normalized MSE learning curves with 95% bootstrap bands for $L = 16$. Bottom: matching MSD curves.

4.4 Time per Update

Figure 5 and Table 5 give the measured time per update. From $L = 4$ to $L = 128$, the median time grew from 0.17 to 0.24 μs for LMS, from 0.22 to 0.30 μs for NLMS, and from 0.61 to 26.27 μs for RLS. Over that range the RLS operation count grows by a factor of about 830, while its measured time grew by a factor of about 43. For small filters the fixed cost of each MATLAB loop iteration hides the difference in operation counts. These times depend on MATLAB and on this computer, so they should not be read as general hardware benchmarks. Initialization took about 19.46 μs for RLS at $L = 128$, small next to a full run.

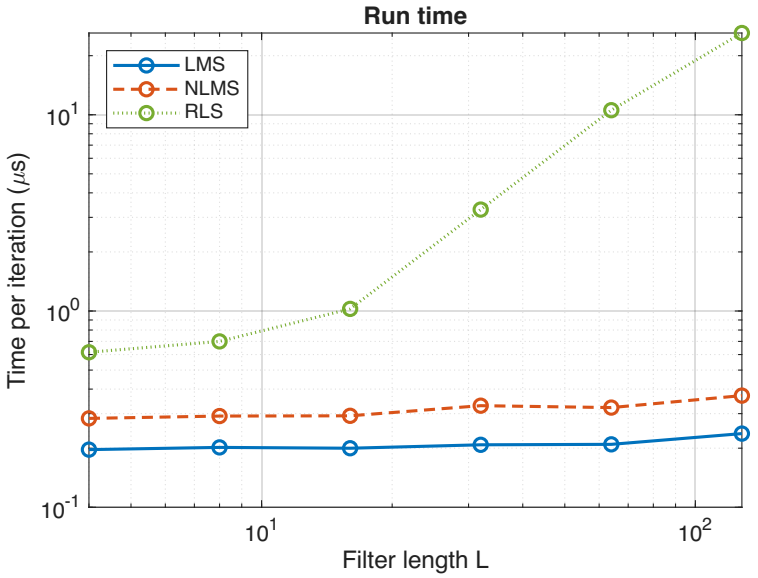

Figure 5: Median measured time per update versus filter length. Error bars show the interquartile range over 15 timed runs.

Table 5: Measured time per update: median (interquartile range).

L	LMS (μs)	NLMS (μs)	RLS (μs)
4	0.20	0.28	0.62
8	0.20	0.29	0.70
16	0.20	0.29	1.03
32	0.21	0.33	3.29
64	0.21	0.32	10.55
128	0.24	0.37	26.09

4.5 Check Against LMS Theory

Figure 6 and Table 6 compare the simulated LMS steady-state MSE with Eq. (12). The biggest gap between simulation and theory was 0.018 dB, and the theory value landed inside the 95% confidence interval of the simulation for 5 of the 5 step sizes. The intervals are about 0.041 dB wide, so the simulation can only confirm agreement to about that level. That is enough to show the LMS code behaves as expected in the white-input, small-μ case where the theory applies.

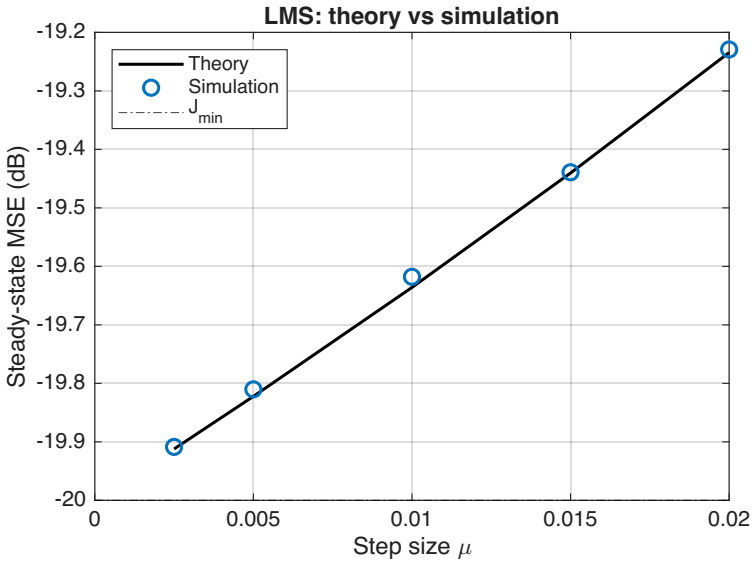

Figure 6: Simulated (with 95% confidence intervals) and theoretical steady-state LMS MSE for white input and $L = 16$.

Table 6: LMS steady-state MSE from simulation and from Eq. (12).

μ	Simulated (dB)	Theory (dB)	Error (dB)
0.0025	-19.91	-19.91	0.00
0.0050	-19.81	-19.82	0.01
0.0100	-19.62	-19.64	0.02
0.0150	-19.44	-19.44	0.00
0.0200	-19.23	-19.23	0.01

4.6 Tracking a Changing System

Figure 7 and Table 7 give the steady-state MSE of each tuned algorithm on fresh test trials while the system drifts. Settings at the edge of the tuning grid carry an asterisk; there were 0 of those.

At every drift level σ_q^2 I tested, LMS, NLMS, and RLS did about the same, with nearly identical steady-state MSE. The differences were small (often under 0.1 dB) and the confidence intervals overlapped, so I cannot call any method better for this scenario. The tuned parameters do tell a clear story, though. As the drift grows from 10^{-9} to 10^{-5}, the selected settings move toward faster adaptation to keep up with the moving target: the best LMS step size μ goes from 0.002 to 0.02, the NLMS step size from 0.05 to 0.5, and the RLS forgetting factor λ from 0.999 to 0.98.

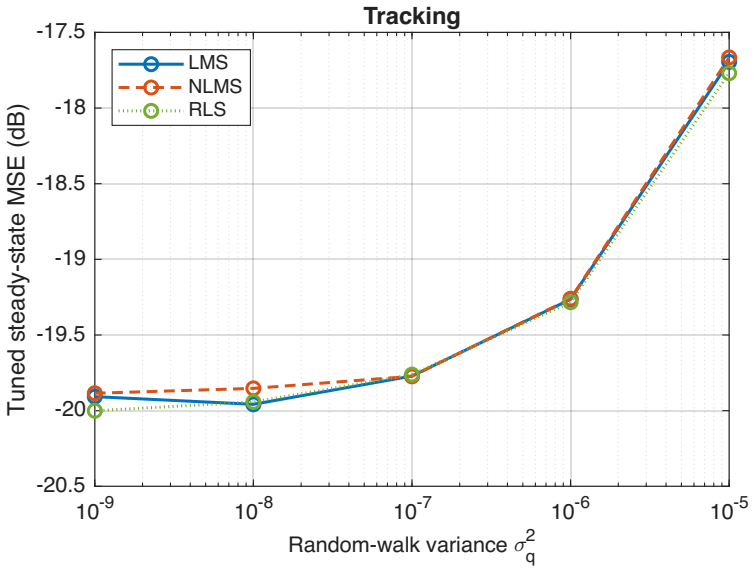

Figure 7: Steady-state MSE of tuned filters versus per-coefficient drift σ_q^2, on test trials. Error bars are 95% bootstrap confidence intervals.

Table 7: Tracking results on test trials: MSE in dB, [selected parameter], and (95% confidence interval). * = selected value is at the edge of the tested grid.

σ_q^2	LMS (dB) [μ]	NLMS (dB) [μ]	RLS (dB) [λ]
10^{-9}	-19.91 [0.002]	-19.88 [0.05]	-20.00 [0.999]
10^{-8}	-19.96 [0.002]	-19.85 [0.05]	-19.94 [0.999]
10^{-7}	-19.77 [0.002]	-19.77 [0.05]	-19.76 [0.995]
10^{-6}	-19.26 [0.01]	-19.26 [0.2]	-19.28 [0.99]
10^{-5}	-17.69 [0.02]	-17.67 [0.5]	-17.77 [0.98]

5 Discussion

Why the winner changes. C_{target} is a product of two factors that pull in opposite directions. RLS needs far fewer iterations, especially for correlated input, since it undoes the eigenvalue spread of $\mathbf{R}$, but its cost per update grows like L^2. LMS and NLMS are cheap per update, yet the number of iterations they need grows with the eigenvalue spread. Neither factor wins everywhere. Whichever one is bigger in a given condition decides the lowest-cost method.

That explains the regime map in Figure 2. With white input the eigenvalue spread is small, so LMS converges quickly and the $\mathcal{O}(L^2)$ cost of RLS buys nothing. As correlation rises LMS needs a lot more iterations, and RLS starts winning in spite of the higher cost per update. The crossover points mostly

match my hypothesis: RLS wins for short filters and strongly correlated inputs, its L^2 scaling makes it lose on long filters. Stronger correlation also pushes the crossover toward longer filters. At $\rho = 0.5$ RLS only tied LMS for $L \leq 16$, while at $\rho = 0.99$ it was the cheapest at every length I tested, up to $L = 128$. If a crossover exists at $\rho = 0.99$, it sits beyond that.

Limitations. These results are bounded by the choices I made:

- Only Gaussian AR(1) inputs, a static system in the main experiment, and a fixed 20-dB output SNR were tested. Real signals such as speech have other correlation patterns.
- The results are for $T = 1.10$ and $T = 1.25$ and a 20000-iteration horizon. "Not reached" means not reached within that horizon, and a longer run might get there.
- Each algorithm was tuned over a seven-point grid of its main parameter, with RLS δ and NLMS ε held fixed. A finer grid could change close results.
- Measured times come from straightforward MATLAB code on one computer. A compiled implementation, a fast RLS variant with $\mathcal{O}(L)$ cost [5], or different hardware could move the crossover. The operation-count results are there to show how much the conclusions depend on this.
- C_{target} measures computation, not real-time delay.

6 Conclusion

I asked when RLS uses less total computation than tuned LMS and NLMS to reach a shared accuracy target. For white or mildly correlated inputs LMS was usually the cheapest. At $\rho \geq 0.9$ RLS needed much less total computation, because it converges in far fewer iterations and that outweighs its steep $\mathcal{O}(L^2)$ cost per update. Longer filters tend to favor LMS and stronger correlation pushes the advantage back to RLS, so the crossover depends on both. The map held up under a different accuracy target and mostly held up when I counted operations instead of timing MATLAB. The biggest limit is that everything here uses synthetic Gaussian AR(1) input and a static unknown system.

Future work could try real recorded signals, a time-varying system in the main sweep, fast RLS variants, and compiled implementations on embedded hardware.

Code and Data Availability

All MATLAB code, settings, seeds, and raw results needed to recreate every figure, table, and number in this paper are available at https://github.com/praneelpa/adaptive-filter-tradeoffs. Running `main.m` once produces all of them.

Acknowledgements

The author thanks Rossi, G.D. for feedback on an earlier draft of this paper.

References

[1] Paulo S. R. Diniz. *Adaptive Filtering: Algorithms and Practical Implementation*. Springer, 4th edition, 2013.

[2] Bradley Efron and Robert J. Tibshirani. *An Introduction to the Bootstrap*. Chapman & Hall, 1993.

[3] Evangelos Eleftheriou and David D. Falconer. Tracking properties and steady-state performance of RLS adaptive filter algorithms. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 34 (5):1097–1110, 1986. doi: 10.1109/TASSP.1986.1164950.

[4] Eweda Eweda. Comparison of RLS, LMS, and sign algorithms for tracking randomly time-varying channels. *IEEE Transactions on Signal Processing*, 42(11):2937–2944, 1994.

[5] Simon Haykin. *Adaptive Filter Theory*. Pearson, 5th edition, 2014.

[6] Ali H. Sayed. *Adaptive Filters*. Wiley, 2008.

[7] Dirk T. M. Slock. On the convergence behavior of the LMS and the normalized LMS algorithms. *IEEE Transactions on Signal Processing*, 41(9):2811–2825, 1993.

[8] Bernard Widrow and Marcian E. Hoff. Adaptive switching circuits. In *IRE WESCON Convention Record, Part 4*, pages 96–104, 1960.

[9] Bernard Widrow, Jr. Glover, John R., John M. McCool, John Kaunitz, Charles S. Williams, Robert H. Hearn, James R. Zeidler, Jr. Dong, Eugene, and Robert C. Goodlin. Adaptive noise cancelling: Principles and applications. *Proceedings of the IEEE*, 63(12):1692–1716, 1975. doi: 10.1109/PROC.1975.10036.

A Tuned Settings

Table 8: Settings selected on the development trials for the main target. * = at the edge of the grid (excluding $\lambda = 1$, which is a natural limit). † = no setting reached the target on the development trials, so the setting with the lowest final error was used.

ρ	L	LMS μL	NLMS μ	RLS $(1-\lambda)L$
0.00	8	0.1	0.1	0
	16	0.1	0.1	0
	32	0.1	0.1	0
	64	0.1	0.1	0
	128	0.1	0.1	0
0.50	8	0.1	0.05	0
	16	0.1	0.1	0
	32	0.1	0.1	0
	64	0.1	0.1	0
	128	0.1	0.1	0
0.90	8	0.1	0.05	0
	16	0.1	0.1	0
	32	0.1	0.1	0
	64	$0.2^{\dagger}$	$0.1^{\dagger}$	0
	128	$0.2^{\dagger}$	$0.2^{\dagger}$	0
0.99	8	$0.1^{\dagger}$	0.05	0
	16	$0.1^{\dagger}$	0.1	0
	32	$0.1^{\dagger}$	$0.1^{\dagger}$	0
	64	$0.1^{\dagger}$	$0.3^{\dagger}$	0
	128	$0.1^{\dagger}$	$0.3^{\dagger}$	0