DSP-CIS

Part-III : Optimal & Adaptive Filters

Chapter-9 :
Square Root and Fast RLS Algorithms

Marc Moonen
Dept. E.E./ESAT-STADIUS, KU Leuven
marc.moonen@kuleuven.be
www.esat.kuleuven.be/stadius/

Part-III : Optimal & Adaptive Filters

Chapter-7 **Optimal Filters - Wiener Filters**
- Introduction : General Set-Up & Applications
- Wiener Filters

Chapter-8 **Adaptive Filters - LMS & RLS**
- Least Means Squares (LMS) Algorithm
- Recursive Least Squares (RLS) Algorithm

Chapter-9 **Square Root & Fast RLS Algorithms**
- Square Root Algorithms
- Fast Algorithms

Chapter-10 **Kalman Filters**
- Introduction – Least Squares Parameter Estimation
- Standard Kalman Filter
- Square-Root Kalman Filter

Recap 1/5

1. Least Squares (LS) Estimation

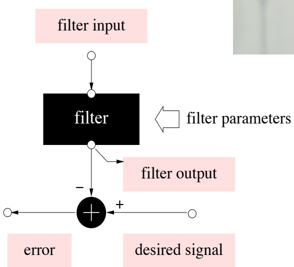
Prototype optimal/adaptive filter revisited

filter structure ?

→ FIR filters
(=pragmatic choice)

cost function ?

→ quadratic cost function
(=pragmatic choice)



Recap 2/5

1. Least Squares (LS) Estimation

Quadratic cost function

MMSE : (see Lecture 8)

$$J_{MSE}(\mathbf{w}) = \mathrm{E}\left\{e_k^2\right\} = \mathrm{E}\left\{(d_k - y_k)^2\right\} = \mathrm{E}\left\{\left(d_k - \mathbf{u}_k^T\mathbf{w}\right)^2\right\}$$

Least-squares(LS) criterion :

if statistical info is not available, may use an alternative 'data-based' criterion...

$$J_{LS}(\mathbf{w}) = \sum_{l=1}^{k} e_l^2 = \sum_{l=1}^{k} (d_l - y_l)^2 = \sum_{l=1}^{k} \left(d_l - \mathbf{u}_l^T\mathbf{w}\right)^2$$

Interpretation? : see below

Recap 3/5

1. Least Squares (LS) Estimation

filter input sequence : $\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3, \ldots \; \mathbf{u}_k$

corresponding desired response sequence is : $d_1, d_2, d_3, \ldots, \; d_k$

$$\underbrace{\begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_k \end{bmatrix}}_{\text{error signal } \mathbf{e}} = \underbrace{\begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_k \end{bmatrix}}_{\mathbf{d}} - \underbrace{\begin{bmatrix} \mathbf{u}_1^T \\ \mathbf{u}_2^T \\ \vdots \\ \mathbf{u}_k^T \end{bmatrix}}_{U} . \underbrace{\begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_L \end{bmatrix}}_{\mathbf{w}}$$

cost function $J_{LS}(\mathbf{w}) = \sum_{l=1}^{k} e_l^2 = \|\mathbf{e}\|_2^2 = \|\mathbf{d} - U\mathbf{w}\|_2^2$

$\rightarrow$ *linear least squares problem* : $\min_{\mathbf{w}} \|\mathbf{d} - U\mathbf{w}\|_2^2$

Recap 4/5

2.1 Standard RLS

It is observed that $\aleph_{uu}[k] = \aleph_{uu}[k-1] + \mathbf{u}_k.\mathbf{u}_k^T$ (and $\aleph_{du}[k] = \aleph_{du}[k-1] + \mathbf{u}_k.d_k$)

The *matrix inversion lemma* states that

$$\aleph_{uu}[k]^{-1} = \aleph_{uu}[k-1]^{-1} - \left(\frac{1}{1 + \mathbf{u}_k^T \aleph_{uu}[k-1]^{-1}\mathbf{u}_k}\right).\mathbf{k}_k\mathbf{k}_k^T \quad \text{with} \quad \mathbf{k}_k = \aleph_{uu}[k-1]^{-1}\mathbf{u}_k$$

$$\mathbf{w}_{LS}[k] = \mathbf{w}_{LS}[k-1] + \underbrace{\overbrace{\aleph_{uu}[k]^{-1}\mathbf{u}_k}^{\text{'Kalman gain vector'}}}_{-\left(\frac{1}{1+\mathbf{u}_k^T \aleph_{uu}[k-1]^{-1}\mathbf{u}_k}\right).\mathbf{k}_k} . \overbrace{(d_k - \mathbf{u}_k^T \mathbf{w}_{LS}[k-1])}^{\text{'a priori residual'}}$$

= **standard recursive least squares (RLS) algorithm**

Remark : $O(L^2)$ instead of $O(L^3)$ *operations per time update*

Recap 5/5

Computational Complexity:

Standard RLS algorithm has **O(L^2)** computational complexity per update

In Chapter-9, will present 'Fast RLS' algorithms with **O(L)** computational complexity

Numerical Analysis/Stability:

Standard RLS algorithm has been shown to have unstable quantization error propagation (in low-precision implementation)

In Chapter-9, will present 'Square-Root RLS' algorithms which are shown to be perfectly stable numerically

Part-III : Optimal & Adaptive Filters

Chapter-7 **Optimal Filters - Wiener Filters**
- Introduction : General Set-Up & Applications
- Wiener Filters

Chapter-8 **Adaptive Filters - LMS & RLS**
- Least Means Squares (LMS) Algorithm
- Recursive Least Squares (RLS) Algorithm

Chapter-9 **Square Root & Fast RLS Algorithms**
- Square Root Algorithms
- Fast Algorithms

Chapter-10 **Kalman Filters**
- Introduction – Least Squares Parameter Estimation
- Standard Kalman Filter
- Square-Root Kalman Filter

Square Root RLS Algorithms

Standard RLS algorithm has been shown to have unstable quantization error propagation (in low-precision implementation)

i.e. when an infinite precision version is run next to a finite precision version (both fed with the same input signals), then after xx iterations the finite precision version produces results (far) away from the infinite precision results

Better ('square root') algorithms are based on orthogonal transformations and 'QR decomposition'

Starting point is again least squares (LS) estimation⋯

Square Root RLS Algorithms

QR Decomposition for LS estimation

least squares problem

$$\min_{\mathbf{w}} \|\mathbf{d} - U\mathbf{w}\|_2^2$$

'square-root algorithms' based on *QR decomposition (QRD)*:

$$\underbrace{U}_{kx(L+1)} = \underbrace{Q}_{kxk} \cdot \underbrace{\begin{bmatrix} R \\ 0 \end{bmatrix}}_{kx(L+1)} = \underbrace{\tilde{Q}}_{Q(:,1:L+1)} \cdot \underbrace{R}_{(L+1)x(L+1)}$$

square * rectangular   rectangular * square

$Q^T \cdot Q = I$, Q is orthogonal R is upper triangular

Everything you need to know about QR decomposition

Example :

$$\overbrace{\begin{bmatrix} 1 & 6 & 10 \\ 2 & 7 & -11 \\ 3 & 8 & 12 \\ 4 & 9 & -13 \end{bmatrix}}^{U} = \overbrace{\begin{bmatrix} 0.182 & 0.816 & 0.174 \\ 0.365 & 0.408 & -0.619 \\ 0.547 & 0 & 0.716 \\ 0.730 & -0.408 & -0.270 \end{bmatrix}}^{\tilde{Q}} \cdot \overbrace{\begin{bmatrix} 5.477 & 14.605 & -5.112 \\ 0 & 4.082 & 8.981 \\ 0 & 0 & 20.668 \end{bmatrix}}^{R}$$

Remark : QRD $\approx$ Gram-Schmidt

Remark : $U^T \cdot U = R^T \cdot R$

R is *Cholesky factor* or *square-root* of $U^T \cdot U$

$\rightarrow$ *'square-root' algorithms* !

Square Root RLS Algorithms

QRD for LS estimation

if

$$\underbrace{U}_{k x(L+1)} = \underbrace{Q}_{kxk} \cdot \underbrace{\begin{bmatrix} R \\ 0 \end{bmatrix}}_{kx(L+1)} = \underbrace{\tilde{Q}}_{Q(:,1:L+1)} \cdot \underbrace{R}_{(L+1)x(L+1)}$$

then

$$\min_{\mathbf{w}} \|\mathbf{d} - U\mathbf{w}\|_2^2 \overset{(**)}{=} \min_{\mathbf{w}} \|Q^T(\mathbf{d} - U\mathbf{w})\|_2^2 = \min_{\mathbf{w}} \left\| \begin{bmatrix} \mathbf{z} \\ * \end{bmatrix} - \begin{bmatrix} R \\ 0 \end{bmatrix} \mathbf{w} \right\|_2^2$$

with this

triangular backsubstitution

$$R \cdot \mathbf{w}_{LS} = z \Rightarrow \mathbf{w}_{LS} = R^{-1} \cdot z = [\tilde{Q}^T U]^{-1} \cdot \tilde{Q}^T \mathbf{d}$$

This is a numerically better way of computing the LS solution, better than $\mathbf{w}_{LS} = [U^T U]^{-1} \cdot U^T \mathbf{d}$

(**) orthogonal transformation preserves norm

Square Root RLS Algorithms

PS: This (= QRD + backsubstitution) is also the way Matlab™ solves LS problems (cfr "x=A\b" or "x=mldivide(A,b)")

Now back to
recursive least squares (RLS) estimation…

This will be based on 'recursive QRD', i.e.

'QRD-updating'

Square Root RLS Algorithms

QR-updating for RLS estimation

Assume we have computed the QRD at time *k-1*

$$\left[\begin{array}{cc} R[k-1] & \mathbf{z}[k-1] \end{array}\right] = \tilde{Q}[k-1]^T \cdot \left[\begin{array}{cc} U_{k-1} & \mathbf{d}_{k-1} \end{array}\right]$$

The corresponding LS solution is $\mathbf{w}_{LS}[k-1] = R[k-1]^{-1} \cdot z[k-1]$

Our aim is to update the QRD into

$$\left[\begin{array}{cc} R[k] & \mathbf{z}[k] \end{array}\right] = \tilde{Q}[k]^T \cdot \left[\begin{array}{cc} U_k & \mathbf{d}_k \end{array}\right]$$

and then compute $\mathbf{w}_{LS}[k] = R[k]^{-1} \cdot z[k]$

Square Root RLS Algorithms

QR-updating for RLS estimation

It is proved that the relevant QRD-updating problem is

$$\begin{bmatrix} R[k] & \mathbf{z}[k] \\ 0 \cdots 0 & * \end{bmatrix} = Q[k]^T \cdot \begin{bmatrix} R[k-1] & \mathbf{z}[k-1] \\ \mathbf{u}_k^T & d_k \end{bmatrix}$$

PS: This is based on a QR-factorization as follows:

$$\underbrace{\begin{bmatrix} R[k-1] \\ \mathbf{u}_k^T \end{bmatrix}}_{(L+2)\times(L+1)} = \underbrace{Q[k]}_{(L+2)\times(L+2)} \cdot \underbrace{\begin{bmatrix} R[k] \\ 0 \end{bmatrix}}_{(L+2)\times(L+1)}$$

Square Root RLS Algorithms

QR-updating for RLS estimation

$$\begin{bmatrix} R[k] & \mathbf{z}[k] \\ 0 \cdots 0 & * \end{bmatrix} = Q[k]^T \cdot \begin{bmatrix} R[k-1] & \mathbf{z}[k-1] \\ \mathbf{u}_k^T & d_k \end{bmatrix}$$

$\mathbf{w}_{LS}[k] = R[k]^{-1} \cdot z[k]$ ='triangular backsubstitution'

= *square-root (information matrix) RLS*

Remark . with exponential weighting

$$\begin{bmatrix} R[k] & \mathbf{z}[k] \\ 0 \cdots 0 & * \end{bmatrix} = Q[k]^T \cdot \begin{bmatrix} \lambda \cdot R[k-1] & \lambda \cdot \mathbf{z}[k-1] \\ \mathbf{u}_k^T & d_k \end{bmatrix}$$

Square Root RLS Algorithms

Will now look into the details of how the Q[k] can be constructed

QRD updating

$$\begin{bmatrix} R[k] & \mathbf{z}[k] \\ 0\cdots0 & * \end{bmatrix} = Q[k]^T \cdot \begin{bmatrix} R[k-1] & \mathbf{z}[k-1] \\ \mathbf{u}_k^T & d_k \end{bmatrix}$$

basic tool is **Givens rotation**

$$G_{i,j,\theta} \stackrel{\text{def}}{=} \begin{array}{cc} \begin{array}{ccccc} & i & & j & \\ & \downarrow & & \downarrow & \end{array} & \\ \begin{bmatrix} I_{i-1} & 0 & 0 & 0 & 0 \\ 0 & \cos\theta & 0 & \sin\theta & 0 \\ 0 & 0 & I_{j-i-1} & 0 & 0 \\ 0 & -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 0 & I_{m-j} \end{bmatrix} & \begin{array}{l} \\ \leftarrow i \\ \\ \leftarrow j \\ \\ \end{array} \end{array}$$

Square Root RLS Algorithms

QRD updating

Givens rotation applied to a vector $\tilde{\mathbf{x}} = G_{i,j,\theta} \cdot \mathbf{x}$:

$$\begin{aligned} \tilde{x}_i &= \cos\theta \cdot x_i + \sin\theta \cdot x_j \\ \tilde{x}_j &= -\sin\theta \cdot x_i + \cos\theta \cdot x_j \end{aligned}$$

$\tilde{x}_l = x_l$ for $l \neq i,j$

$\tilde{x}_j = 0$ iff $\tan\theta = \frac{x_j}{x_i}$!

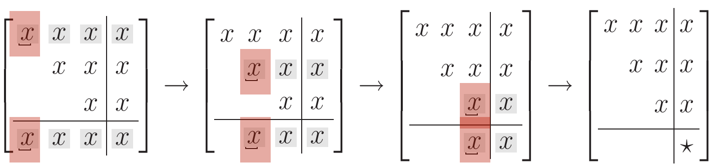
Square Root RLS Algorithms

QRD updating

$$\begin{bmatrix} R[k] & \mathbf{z}[k] \\ 0\cdots0 & * \end{bmatrix} = Q[k]^T \cdot \begin{bmatrix} R[k-1] & \mathbf{z}[k-1] \\ \mathbf{u}_k^T & d_k \end{bmatrix}$$

$Q[k]$ is constructed as a product/sequence of Givens transformations

3-by-3 example



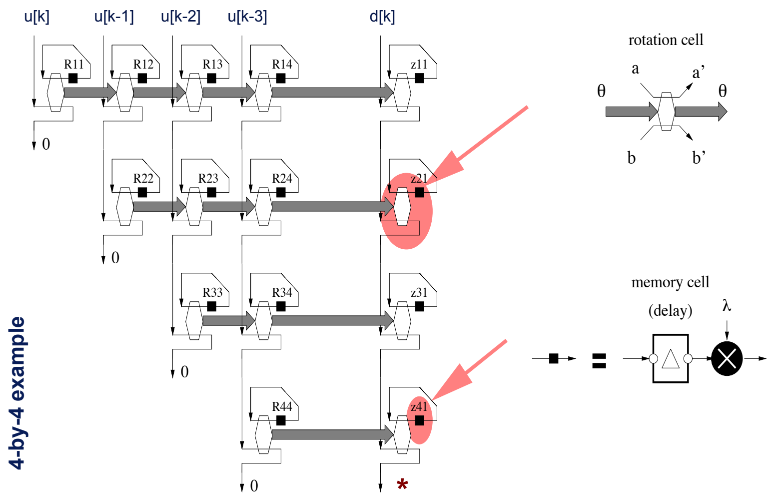
Square Root RLS Algorithms

A graphical representation (i.e. a ‘realization’) of the QRD updating process is presented in the next slide

This is also referred to as a ‘signal flow graph’ (SFG)

The SFG in the next slide will be further developed in later slides, and also used explicitly for the (graphical) derivation of a ‘fast’ RLS algorithm (p.27)

$$\begin{bmatrix} R[k] & \mathbf{z}[k] \\ 0\cdots 0 & * \end{bmatrix} = Q[k]^T \cdot \begin{bmatrix} R[k-1] & \mathbf{z}[k-1] \\ \mathbf{u}_k^T & d_k \end{bmatrix}$$



Square Root RLS Algorithms

Residual Extraction

So far, the "star" (*) in the update equation
(also appearing at the bottom of the SFG)
has not been considered/defined/used

It will turn out that this "star" can be used to compute least squares residuals
(without explicitly computing the least squares filter vector!)

Square Root RLS Algorithms

Residual extraction

$$\begin{bmatrix} R[k] & \mathbf{z}[k] \\ 0\cdots 0 & \boxed{\varepsilon} \end{bmatrix} = Q[k]^T \cdot \begin{bmatrix} R[k-1] & \mathbf{z}[k-1] \\ \mathbf{u}_k^T & d_k \end{bmatrix}$$

From this it is proved that the 'a posteriori residual' is

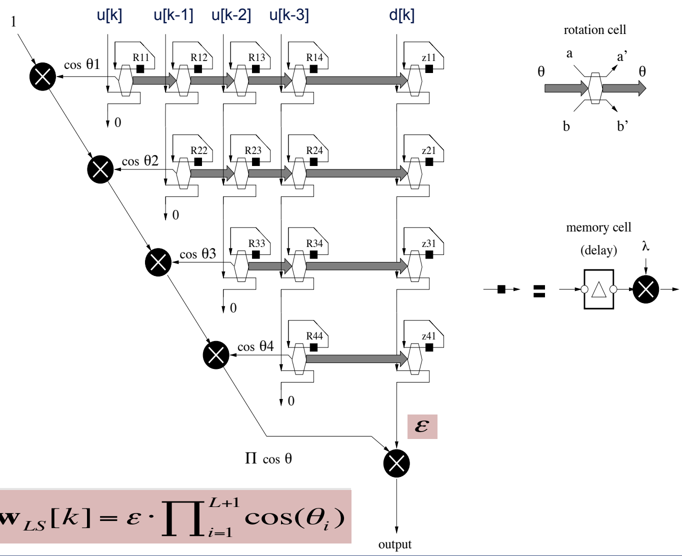
$$d_k - \mathbf{u}_k^T \mathbf{w}_{LS}[k] = \varepsilon \cdot \prod_{i=1}^{L+1} \cos(\theta_i)$$

and the 'a priori residual' is

$$d_k - \mathbf{u}_k^T \mathbf{w}_{LS}[k-1] = \frac{\varepsilon}{\prod_{i=1}^{L+1} \cos(\theta_i)}$$

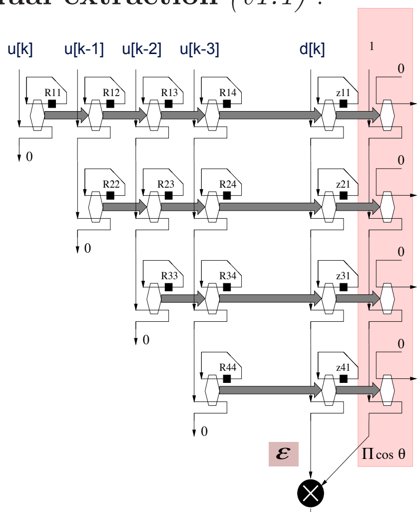
Hence ε is geometric mean of a posteriori & a priori residual

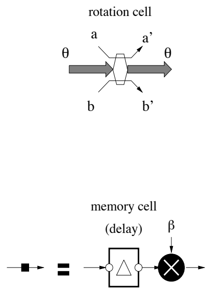
Residual extraction *(v1.0)* :



$$d_k - \mathbf{u}_k^T \mathbf{w}_{LS}[k] = \varepsilon \cdot \prod_{i=1}^{L+1} \cos(\theta_i)$$

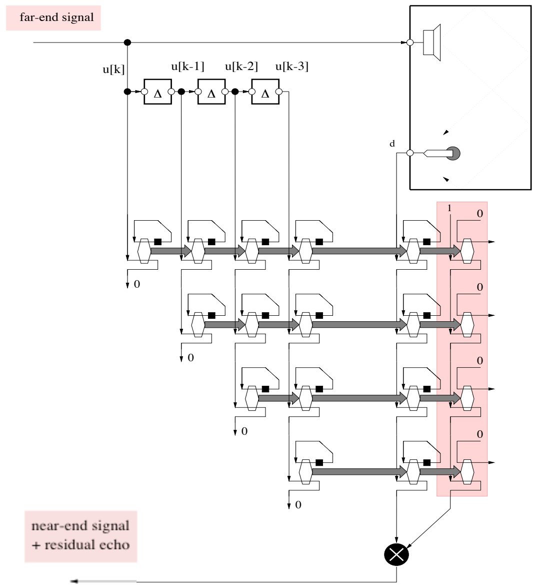
Residual extraction *(v1.1)* :





$$d_k - \mathbf{u}_k^T \mathbf{w}_{LS}[k] = \varepsilon \cdot \prod_{i=1}^{L+1} \cos(\theta_i)$$

Example



Part-III : Optimal & Adaptive Filters

Chapter-7 **Optimal Filters - Wiener Filters**
- Introduction : General Set-Up & Applications
- Wiener Filters

Chapter-8 **Adaptive Filters - LMS & RLS**
- Least Means Squares (LMS) Algorithm
- Recursive Least Squares (RLS) Algorithm

Chapter-9 **Square Root & Fast RLS Algorithms**
- Square Root Algorithms
- Fast Algorithms

Chapter-10 **Kalman Filters**
- Introduction – Least Squares Parameter Estimation
- Standard Kalman Filter
- Square-Root Kalman Filter

Fast RLS Algorithms

RLS and square-root RLS : $O(L^2)$ per time update

When the adaptive filter is an FIR filter, the computational cost may be reduced to $O(L)$ per time update, by exploiting the time-shift structure of the input vectors/signals !

Will consider/derive 1 'fast' algorithm here:

- **QRD least squares lattice (QRD-LSL)**

Other :

- **Least-squares lattice (LSL)**
- **'Fast QR'**
- **Fast transversal filter (FTF)**

Fast RLS Algorithms

Preliminaries

- vast literature available on *fast least squares algorithms*
- the derivation of fast algorithms is *highly* mathematical (see page 32)
- we show how fast (QRD-based) algorithms can be derived using *signal flow graph (SFG) manipulation*
- In doing so we provide additional insight to the algorithmic structure

Fast RLS Algorithms

Example (headache?)

See p.39 for a signal flow graph of this

9.2.1. QRD-based Least Squares Lattice algorithm.

START

INITIALISE {all variables} := 0;

FOR n FROM 1 DO

LET $\alpha_{f,0}(n) := x(n)$; $\alpha_{b,0}(n-1) := x(n-1)$; $\alpha_0(n-1) := y(n-1)$; $\gamma_0(n-1) := 1$;

FOR q FROM 1 TO p DO

LET $\varepsilon_{b,q-1}(n-1) := \sqrt{(\beta\varepsilon_{b,q-1}(n-2))^2 + |\alpha_{b,q-1}(n-1)|^2}$;

IF $\varepsilon_{b,q-1}(n-1) = 0$ THEN LET $c_{f,q} := 1$; $s_{f,q} := 0$

ELSE LET $c_{f,q} := \beta\varepsilon_{b,q-1}(n-2) / \varepsilon_{b,q-1}(n-1)$; $s_{f,q} := \alpha_{b,q-1}(n-1) / \varepsilon_{b,q-1}(n-1)$

END_IF;

LET $\mu_{f,q-1}(n) := c_{f,q}\,\beta\mu_{f,q-1}(n-1) + s^*_{f,q}\,\alpha_{f,q-1}(n)$;

$\alpha_{f,q}(n) := c_{f,q}\,\alpha_{f,q-1}(n) - s_{f,q}\,\beta\mu_{f,q-1}(n-1)$;

$\mu_{q-1}(n-1) := c_{f,q}\,\beta\mu_{q-1}(n-2) + s^*_{f,q}\,\alpha_{q-1}(n-1)$;

$\alpha_q(n-1) := c_{f,q}\,\alpha_{q-1}(n-1) - s_{f,q}\,\beta\mu_{q-1}(n-2)$;

$\gamma_q(n-1) := c_{f,q}\,\gamma_{q-1}(n-1)$;

COMMENT prediction residual $e_{f,p}(n,n) = \gamma_q(n-1)\,\alpha_{f,q}(n)$ COMMENT

$e_p(n-1,n-1) = \gamma_q(n-1)\,\alpha_q(n-1)$ COMMENT q-th order filtered residual COMMENT

LET $\varepsilon_{f,q-1}(n) := \sqrt{(\beta\varepsilon_{f,q-1}(n-1))^2 + |\alpha_{f,q-1}(n)|^2}$;

IF $\varepsilon_{f,q-1}(n) = 0$ THEN LET $c_{b,q} := 1$; $s_{b,q} := 0$

ELSE LET $c_{b,q} := \beta\varepsilon_{f,q-1}(n-1) / \varepsilon_{f,q-1}(n)$; $s_{b,q} := \alpha_{f,q-1}(n) / \varepsilon_{f,q-1}(n)$

END_IF;

LET $\mu_{b,q-1}(n-1) := c_{b,q}\,\beta\mu_{b,q-1}(n-2) + s^*_{b,q}\,\alpha_{b,q-1}(n-1)$;

$\alpha_{b,q}(n) := c_{b,q}\,\alpha_{b,q-1}(n-1) - s_{b,q}\,\beta\mu_{b,q-1}(n-2)$;

COMMENT $\gamma_q(n) := c_{b,q}\,\gamma_{q-1}(n-1)$; backward prediction residual $e_{b,q}(n,n) := \gamma_q(n)\,\alpha_{b,q}(n)$ COMMENT

END_DO

END_DO

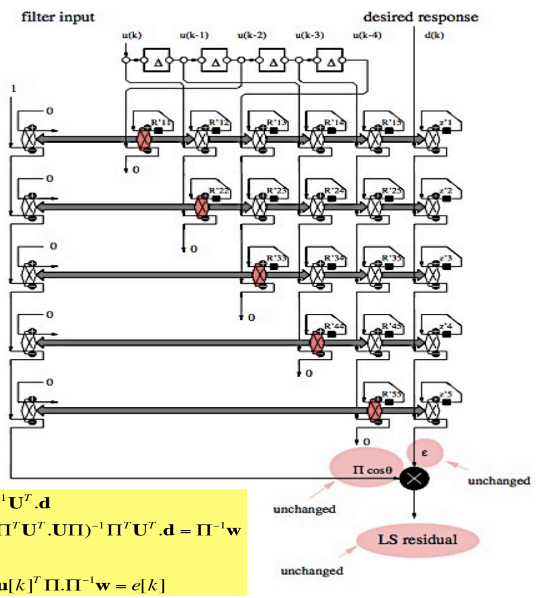
FINISH

Fast RLS Algorithms

Preliminaries

- LS residuals are not changed after a permutation of the input signals (see page 32)
- This allows for a compact notation (ε-notation) for all intermediate signals in the SFG :
  Every intermediate signal corresponds to the 'ε-signal' of an embedded LS problem.
  Then in the ε-notation, the superscript refers to the (time-index of the) 'right-hand side signal' of this LS problem, the subscript (Matlab-like notation) refers to the (time indices of the) *set* of 'left-hand side signals' of this LS problem
  (see page 33)

Preliminaries



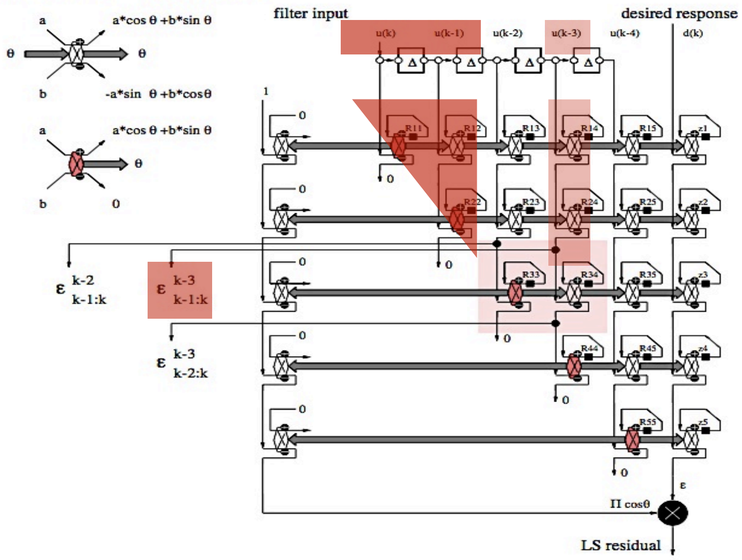
$$\mathbf{w} = \arg\min_{\mathbf{w}} \| \mathbf{d} - \mathbf{U}.\mathbf{w} \|_2^2 = (\mathbf{U}^T.\mathbf{U})^{-1}\mathbf{U}^T.\mathbf{d}$$

$$\Rightarrow \tilde{\mathbf{w}} = \arg\min_{\tilde{\mathbf{w}}} \| \mathbf{d} - (\mathbf{U}\Pi).\tilde{\mathbf{w}} \|_2^2 = (\Pi^T\mathbf{U}^T.\mathbf{U}\Pi)^{-1}\Pi^T\mathbf{U}^T.\mathbf{d} = \Pi^{-1}\mathbf{w}$$

$$e[k] = d[k] - \mathbf{u}[k]^T.\mathbf{w}$$

$$\Rightarrow \tilde{e}[k] = d[k] - \mathbf{u}[k]^T\Pi.\tilde{\mathbf{w}} = d[k] - \mathbf{u}[k]^T\Pi.\Pi^{-1}\mathbf{w} = e[k]$$

Preliminaries : ε-notation

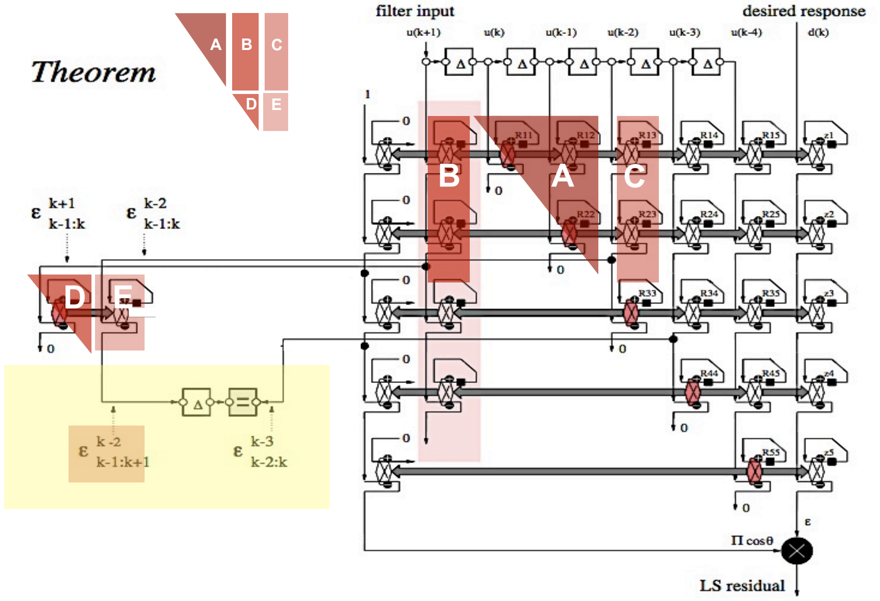


Fast RLS Algorithms

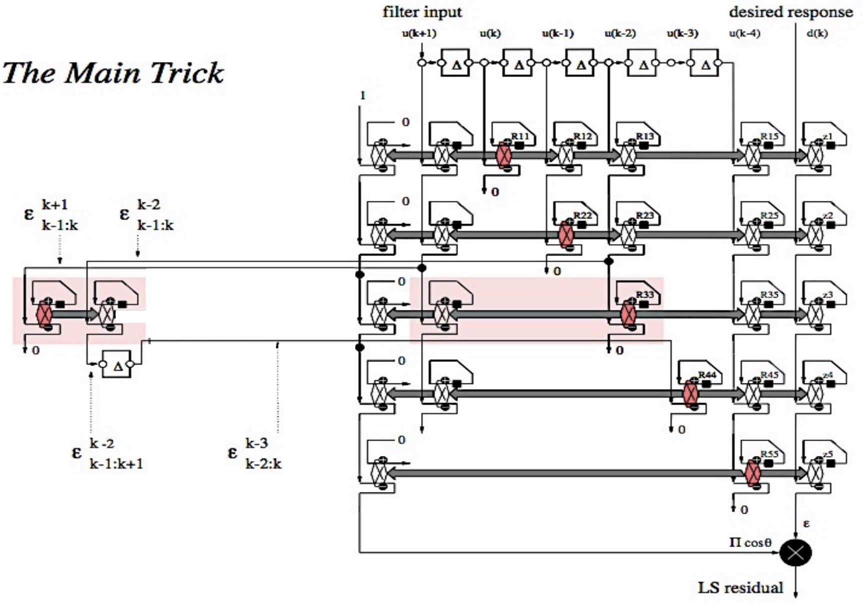
QRD least squares lattice (QRD-LSL)

- SFG derivation (=3 figures)
- Page 35: Try to understand the equation with the ε's
- Page 36: $\varepsilon^{k-3}_{k-2:k}$ is generated in the left-hand side part, hence its original generation (everything above R_{44}) can be left out.
- Page 37 : Apply this trick in each column...

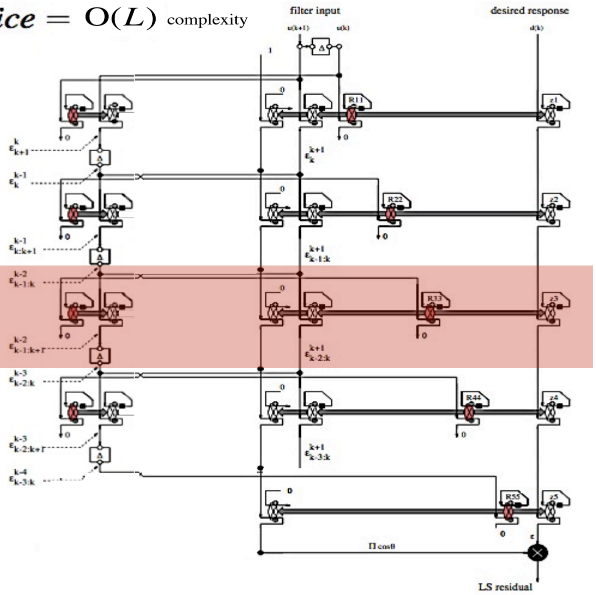
Theorem



The Main Trick



Result = *QRD Lattice* = $O(L)$ complexity



Fast RLS Algorithms

Number of 'layers' = filter order L +1

Six rotations per layer, four multiplications per rotation, hence overall complexity is ≈24(L+1)

(compare to LMS & (standard) RLS)

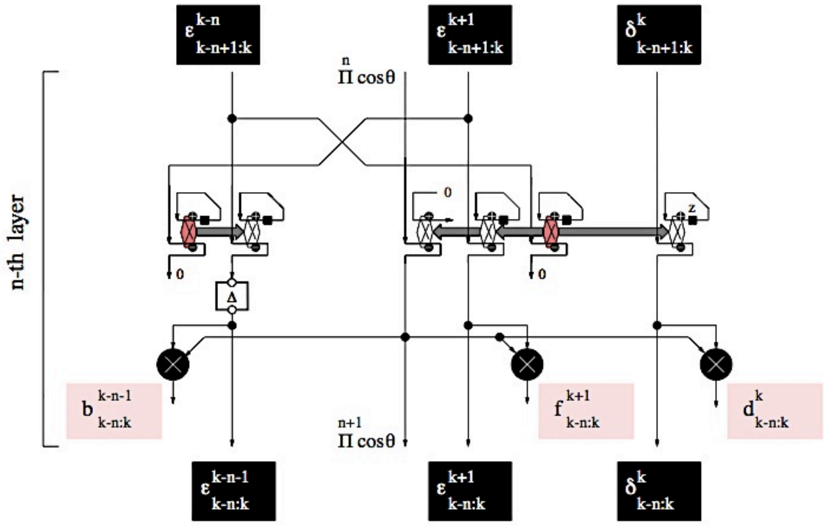
Each layer has the same structure (see next slide)

Epsilon's correspond to forward and backward prediction problems applied to input signal u[k]

By multiplying epsilon's with cosine-products true forward (f) and backward (b) residuals are obtained

Prediction order is increased when going from one layer to the next lower layer

QRD-LSL layer (with forward/backward residuals extraction)



Conclusion

- Many ‘fast’ RLS algorithms available (QRD-lattice, LSL, Fast-QR, FTF,...)
- High performance (*cfr.* RLS) at low cost ($O(L)$, *i.e.* almost as cheap as LMS)
- Derivation is very mathematical...
- ..but SFG’s may help.