Burg's Method, Algorithm and Recursion

Cedrick Collomb
http://ccollomb.free.fr/

Created: November 8, 2009
Last Modified: November 10, 2009

Contents

1. Linear Prediction Recap

Given a discrete set of N original values $(x_n)_{n\in[\![0,N]\!]}$, we use k coefficients $(a_n)_{n\in[\![1,k]\!]}$ to approximate the original values by $y_n = -\sum_{i=1}^{k} a_i x_{n-i}$ for what is called the forward linear prediction, and by $z_n = -\sum_{i=1}^{k} a_i x_{n+i}$ for what is called the backward linear prediction.

Simply put each y_n is a linear weighted combination of the k previous known values and each z_n is a linear weighted combination of the k next known values. Therefore note that y_n is only defined for $n \in [\![k, N]\!]$, and z_n is only

defined for $n \in [\![0, N-k]\!]$.

The usual way to chose $(a_n)_{n\in[\![1,k]\!]}$ is by minimizing the sum of the squares of the error between the original and approximated values for example for the forward linear prediction we would try to minimize F_k in the following formula.

$$F_k = \sum_{n=k}^{N} (x_n - y_n)^2 = \sum_{n=k}^{N} \left(x_n - \left(-\sum_{i=1}^{k} a_i x_{n-i} \right) \right)^2 \quad (1)$$

For the backward linear prediction we would try to minimize B_k in the following formula.

$$B_k = \sum_{n=k}^{N} (x_n - z_n)^2 = \sum_{n=0}^{N-k} \left(x_n - \left(-\sum_{i=1}^{k} a_i x_{n+i} \right) \right)^2 \quad (2)$$

The issue with solving for $(a_n)_{n\in[\![1,k]\!]}$ by minimizing either F_k or B_k as described for example in Collomb (2009), is that the method uses covariance or autocorrelation coefficients which are ill suited for numerical computation, but also because the models are not always stable models (Hayes, 2002). What does that mean in practice? It simply means that the $(a_n)_{n\in[\![1,k]\!]}$ returned by the algorithm fail to be useful and do not approximate well the original values. Therefore, a more robust and stable solution is desirable, and that is exactly what the Burg's method is.

2. Burg's Method

a. The very simple idea

Burg's idea is remarkably simple but is either not explained (Press et al., 2002), sometimes hidden behind unnecessary obfuscations and vocabulary (Burg, 1975), sometimes not rigorously derived (Claerbout, 1997), or often buried under the unnecessary whole lattice filters theory (Hayes, 2002).

Imagine yourself in 1975, you are using the Levinson-Durbin recursion (Collomb, 2009), ideally you would like to find a solution with similar computation requirements but without the instability. That is exactly what Burg has done by reusing the Levinson-Durbin recursion with a different constraint.

In the original Levinson-Durbin recursion, the coefficients $(a_n)_{n\in[\![1,k]\!]}$ are

stored in a vector $A_k = \begin{bmatrix} 1 \\ a_1 \\ a_2 \\ \vdots \\ a_k \end{bmatrix}$ and an inverted order vector $V_k = \begin{bmatrix} 0 \\ a_k \\ \vdots \\ a_2 \\ a_1 \\ 1 \end{bmatrix}$.

The recursion formula is therefore the following

$$A_{k+1} = A_k + \mu V_k \tag{3}$$

With μ computed so that to respect the initial problem conditions as detailed in Collomb (2009). Burg's idea was simply to change the way μ is computed, so that not to fit the initial problem conditions, but to instead minimize the total sum of $F_k + B_k$ introduced in (1) and (2).

That is it, that was not too complicated was it? Now it is time to derive the formulas.

b. Some preparative notations

Before going further, it is better to prepare by changing few notations to make the next steps easier. Reworking equation (1) by defining $a_0 = 1$ gives

$$F_k = \sum_{n=k}^{N} \left(a_0 x_n + \sum_{i=1}^{k} a_i x_{n-i} \right)^2 = \sum_{n=k}^{N} \left(\sum_{i=0}^{k} a_i x_{n-i} \right)^2 = \sum_{n=k}^{N} \left(f_k(n) \right)^2 \tag{4}$$

With

$$f_k(n) = \sum_{i=0}^{k} a_i x_{n-i} \tag{5}$$

Similarly, reworking equation (2) gives

$$B_k = \sum_{n=0}^{N-k} \left(a_0 x_n + \sum_{i=1}^{k} a_i x_{n+i} \right)^2 = \sum_{n=0}^{N-k} \left(\sum_{i=0}^{k} a_i x_{n-i} \right)^2 = \sum_{n=0}^{N-k} \left(b_k(n) \right)^2 \tag{6}$$

With

$$b_k(n) = \sum_{i=0}^{k} a_i x_{n+i} \tag{7}$$

Finally writing A_{k+1} as the vector of the coefficients $\left(a'_n\right)_{n \in [\![1,k+1]\!]}$, using (3) and the fact that V_k is simply the inverted of A_k, and defining $a_{k+1} = 0$, we get

$$a_n^{'} = a_n + \mu a_{k+1-n} \tag{8}$$

c. Solving for μ

Assuming that we have found A_k, in order to find μ, we need to use (4) and (6), and simply need to minimize

$$F_{k+1} + B_{k+1} = \sum_{n=k+1}^{N} \left(f_{k+1}(n) \right)^2 + \sum_{n=0}^{N-k-1} \left(b_{k+1}(n) \right)^2 \tag{9}$$

From (5) and (8) we derive

$$\begin{aligned} f_{k+1}(n) &= \sum_{i=0}^{k+1} a_i^{'} x_{n-i} \\ &= \sum_{i=0}^{k+1} a_i \, x_{n-i} + \mu \sum_{i=0}^{k+1} a_{k+1-i} x_{n-i} \\ &= f_k(n) + \mu \sum_{j=0}^{k+1} a_j x_{n-k-1+j} \end{aligned}$$

Therefore, $$f_{k+1}(n) = f_k(n) + \mu b_k(n-k-1) \tag{10}$$

From (7) and (8) we derive

$$\begin{aligned} b_{k+1}(n) &= \sum_{i=0}^{k+1} a_i^{'} x_{n+i} \\ &= \sum_{i=0}^{k+1} a_i \, x_{n+i} + \mu \sum_{i=0}^{k+1} a_{k+1-i} x_{n+i} \\ &= b_k(n) + \mu \sum_{j=0}^{k+1} a_j x_{n+k+1-j} \end{aligned}$$

Therefore, $$b_{k+1}(n) = b_k(n) + \mu f_k(n+k+1) \tag{11}$$

Plugging (10) and (11) into (9) gives

$$F_{k+1} + B_{k+1} = \sum_{n=k+1}^{N} \left(f_k(n) + \mu b_k(n-k-1) \right)^2 + \sum_{n=0}^{N-k-1} \left(b_k(n) + \mu f_k(n+k+1) \right)^2 \tag{12}$$

This can be minimized by simply finding when the derivative of the μ variable is zero. Therefore, we need to find μ so that

$$\frac{\partial\left(F_{k+1}+B_{k+1}\right)}{\partial\mu}=0 \tag{13}$$

Plugging (12) into (13) we get

$$\begin{aligned}0&=\frac{\partial\left(\sum_{n=k+1}^{N}\left(f_k(n)+\mu b_k(n-k-1)\right)^2+\sum_{n=0}^{N-k-1}\left(b_k(n)+\mu f_k(n+k+1)\right)^2\right)}{\partial\mu}\\&=2\sum_{n=k+1}^{N}b_k(n-k-1)\left(f_k(n)+\mu b_k(n-k-1)\right)\\&\quad+2\sum_{n=0}^{N-k-1}f_k(n+k+1)\left(b_k(n)+\mu f_k(n+k+1)\right)\end{aligned}$$

Therefore

$$\begin{aligned}0&=\sum_{n=k+1}^{N}b_k(n-k-1)\left(f_k(n)+\mu b_k(n-k-1)\right)\\&\quad+\sum_{n=0}^{N-k-1}f_k(n+k+1)\left(b_k(n)+\mu f_k(n+k+1)\right)\end{aligned}$$

Thus

$$\begin{aligned}0&=\sum_{n=k+1}^{N}f_k(n)b_k(n-k-1)+\mu\sum_{n=k+1}^{N}b_k(n-k-1)^2\\&\quad+\sum_{n=0}^{N-k-1}f_k(n+k+1)b_k(n)+\mu\sum_{n=0}^{N-k-1}f_k(n+k+1)^2\end{aligned}$$

Extracting μ is now easy and gives

$$\mu=-\frac{\sum_{n=k+1}^{N}f_k(n)b_k(n-k-1)+\sum_{n=0}^{N-k-1}f_k(n+k+1)b_k(n)}{\sum_{n=0}^{N-k-1}f_k(n+k+1)^2+\sum_{n=k+1}^{N}b_k(n-k-1)^2}$$

Simplifying by adjusting indices and bounds results in

$$\mu=\frac{-2\sum_{n=0}^{N-k-1}f_k(n+k+1)b_k(n)}{\sum_{n=k+1}^{N}f_k(n)^2+\sum_{n=0}^{N-k-1}b_k(n)^2} \tag{14}$$

3. Burg's Algorithm

a. First pass algorithm

At this stage we have enough to implement a first pass version of the algorithm following the steps below.

i. Choose m, the number of wanted coefficients
ii. Initialize $A_0 = [1]$
iii. Using (5) and (7), initialize all $f_0(n) = b_0(n) = x_n$
iv. For k from 0 to m - 1
- Calculate μ using (14)
- Update A_{k+1} using (8)
- Update $\left(f_{k+1}(n)\right)_{n\in[\![k+1,N]\!]}$ using (5)
- Update $\left(b_{k+1}(n)\right)_{n\in[\![0,N-k-1]\!]}$ using (7)

b. Improved algorithm

The previous algorithm is fine except that the updates of $\left(f_{k+1}(n)\right)_{n\in[\![k+1,N]\!]}$ and $\left(b_{k+1}(n)\right)_{n\in[\![0,N-k-1]\!]}$ require computing sums which are not necessary. The recursive formulas (10) and (11) can be used to get to the same result quicker and with less effort.

i. Choose m, the number of wanted coefficients
ii. Initialize $A_0 = [1]$
iii. Using (5) and (7), initialize all $f_0(n) = b_0(n) = x_n$
iv. For k from 0 to m - 1
- Calculate μ using (14)
- Update A_{k+1} using (8)
- Update $\left(f_{k+1}(n)\right)_{n\in[\![k+1,N]\!]}$ using (10)
- Update $\left(b_{k+1}(n)\right)_{n\in[\![0,N-k-1]\!]}$ using (11)

4. Burg's recursion

a. The derivation

The improved algorithm is better, however there are still some simplifications that can happen. Calculating μ can be done more simply and lead to an improved third version referred as Burg's recursion in the literature.

First of all note that the denominator D_k of μ is

$$D_k = F_k - f_k(k)^2 + B_k - b_k(N-k)^2 \tag{15}$$

And we would like to find a recursive formula so that to be able to calculate $D_{k+1} = F_{k+1} - f_{k+1}(k+1)^2 + B_{k+1} - b_{k+1}(N-k-1)^2$. Expanding the squares in (12) and collecting around μ we find

$$\begin{aligned}
F_{k+1} + B_{k+1} &= \sum_{n=k+1}^{N} \left(f_k(n)^2 + 2\mu b_k(n-k-1) f_k(n) + \mu^2 b_k(n-k-1)^2 \right) \\
&\quad + \sum_{n=0}^{N-k-1} \left(b_k(n)^2 + 2\mu b_k(n) f_k(n+k+1) + \mu^2 f_k(n+k+1)^2 \right) \\
&= \sum_{n=k+1}^{N} f_k(n)^2 + \sum_{n=0}^{N-k-1} b_k(n)^2 \\
&\quad + 2\mu \left(\sum_{n=k+1}^{N} b_k(n-k-1) f_k(n) + \sum_{n=0}^{N-k-1} b_k(n) f_k(n+k+1) \right) \\
&\quad + \mu^2 \left(\sum_{n=k+1}^{N} b_k(n-k-1)^2 + \sum_{n=0}^{N-k-1} f_k(n+k+1)^2 \right)
\end{aligned}$$

Using (4) and (6), and updating indices we get

$$\begin{aligned}
F_{k+1} + B_{k+1} &= F_k - f_k(k)^2 + B_k - b_k(N-k)^2 \\
&\quad + 2\mu \left(\sum_{j=0}^{N-k-1} b_k(j) f_k(j+k+1) + \sum_{n=0}^{N-k-1} b_k(n) f_k(n+k+1) \right) \\
&\quad + \mu^2 \left(\sum_{n=0}^{N-k-1} b_k(n)^2 + \sum_{n=k+1}^{N} f_k(n)^2 \right)
\end{aligned}$$

Therefore

$$
\begin{aligned}
F_{k+1}+B_{k+1} &= F_k - f_k(k)^2 + B_k - b_k(N-k)^2 \\
&\quad +4\mu\left(\sum_{n=0}^{N-k-1} b_k(n)\, f_k(n+k+1)\right) \\
&\quad +\mu^2\left(F_k - f_k(k)^2 + B_k - b_k(N-k)^2\right)
\end{aligned}
$$

Factorizing and using (14) gives

$$
\begin{aligned}
F_{k+1}+B_{k+1} &= \left(1+\mu^2\right)\left(F_k - f_k(k)^2 + B_k - b_k(N-k)^2\right) \\
&\quad +4\mu\left(-\frac{\mu}{2}\left(\sum_{n=k+1}^{N} f_k(n)^2 + \sum_{n=0}^{N-k-1} b_k(n)^2\right)\right) \\
&= \left(1+\mu^2\right)\left(F_k - f_k(k)^2 + B_k - b_k(N-k)^2\right) \\
&\quad -2\mu^2\left(F_k - f_k(k)^2 + B_k - b_k(N-k)^2\right) \\
&= \left(1-\mu^2\right)\left(F_k - f_k(k)^2 + B_k - b_k(N-k)^2\right) \\
&= \left(1-\mu^2\right) D_k
\end{aligned}
$$

And finally

$$
D_{k+1} = \left(1-\mu^2\right) D_k - f_{k+1}(k+1)^2 - b_{k+1}(N-k-1)^2 \qquad (16)
$$

b. The final algorithm

i. Choose m, the number of wanted coefficients
ii. Initialize $A_0 = [1]$
iii. Using (5) and (7), initialize all $f_0(n) = b_0(n) = x_n$
iv. Using (4) and (6), compute F_0 and B_0
v. Using (15) compute D_0
vi. For k from 0 to m - 1

- Calculate μ using $\mu = \dfrac{-2\sum_{n=0}^{N-k-1} f_k(n+k+1)\, b_k(n)}{D_k}$
- Update A_{k+1} using (8)
- Update $\left(f_{k+1}(n)\right)_{n\in[\![k+1,N]\!]}$ using (10)
- Update $\left(b_{k+1}(n)\right)_{n\in[\![0,N-k-1]\!]}$ using (11)
- Update D_k using (16)

5. References

Burg, J.P. (1975). Maximum Entropy Spectral Analysis. Available online at http://sepwww.stanford.edu/theses/sep06/

Claerbout, J. (1997). Burg Spectral Estimation. Available online at http://sepwww.stanford.edu/sep/prof/fgdp/c7/paper_html/node3.html

Collomb, C. (2009). Linear Prediction and Levinson-Durbin Algorithm. Available online at http://ccollomb.free.fr/technotes/A%20tutorial%20on%20linear%20prediction%20and%20Levinson-Durbin.pdf

Hayes, M.H. (2002). Statistical Digital Signal Processing and Modeling. Wileys & Sons.

Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P. (2002). Numerical Recipes in C++. Cambridge University Press.

6. Appendix. Non optimized C++ code

```
#include <math.h>
#include <vector>

using namespace std;

// Returns in vector coefficients calculated using Burg algorithm applied to the input source data x

void BurgAlgorithm( vector<double> &coeffs, const vector<double> &x )
{
	// GET SIZE FROM INPUT VECTORS
	size_t N = x.size() - 1;
	size_t m = coeffs.size();

	// INITIALIZE Ak
	vector<double> Ak( m + 1, 0.0 );
	Ak[ 0 ] = 1.0;

	// INITIALIZE f and b
	vector<double> f( x );
	vector<double> b( x );

	// INITIALIZE Dk
	double Dk = 0.0;
	for ( size_t j = 0; j <= N; j++ )
	{
		Dk += 2.0 * f[ j ] * f[ j ];
```

```
        }
        Dk -= f[ 0 ] * f[ 0 ] + b[ N ] * b[ N ];

        // BURG RECURSION
        for ( size_t k = 0; k < m; k++ )
        {
                // COMPUTE MU
                double mu = 0.0;
                for ( size_t n = 0; n <= N - k - 1; n++ )
                {
                        mu += f[ n + k + 1 ] * b[ n ];
                }
                mu *= -2.0 / Dk;

                // UPDATE Ak
                for ( size_t n = 0; n <= ( k + 1 ) / 2; n++ )
                {
                        double t1 = Ak[ n ] + mu * Ak[ k + 1 - n ];
                        double t2 = Ak[ k + 1 - n ] + mu * Ak[ n ];
                        Ak[ n ] = t1;
                        Ak[ k + 1 - n ] = t2;
                }

                // UPDATE f and b
                for ( size_t n = 0; n <= N - k - 1; n++ )
                {
                        double t1 = f[ n + k + 1 ] + mu * b[ n ];
                        double t2 = b[ n ] + mu * f[ n + k + 1 ];
                        f[ n + k + 1 ] = t1;
                        b[ n ] = t2;
                }

                // UPDATE Dk
                Dk = ( 1.0 - mu * mu ) * Dk - f[ k + 1 ] * f[ k + 1 ] - b[ N - k - 1 ] * b[ N - k - 1 ];
        }

        // ASSIGN COEFFICIENTS
        coeffs.assign( ++Ak.begin(), Ak.end() );
}
```

// Example program using Burg's algorithm

```
int main( int argc, char *argv[] )
{
        // CREATE DATA TO APPROXIMATE
        vector<double> original( 128, 0.0 );
        for ( size_t i = 0; i < original.size(); i++ )
        {
                original[ i ] = cos( i * 0.01 ) + 0.75 *cos( i * 0.03 )
                                + 0.5 *cos( i * 0.05 ) + 0.25 *cos( i * 0.11 );
        }

        // GET LINEAR PREDICTION COEFFICIENTS
        vector<double> coeffs( 4, 0.0 );
        BurgAlgorithm( coeffs, original );


        // LINEAR PREDICT DATA
        vector<double> predicted( original );
        size_t m = coeffs.size();
        for ( size_t i = m; i < predicted.size(); i++ )
        {
                predicted[ i ] = 0.0;
                for ( size_t j = 0; j < m; j++ )
                {
                        predicted[ i ] -= coeffs[ j ] * original[ i - 1 - j ];
                }
```

```
        }

        // CALCULATE AND DISPLAY ERROR
        double error = 0.0;
        for ( size_t i = m; i < predicted.size(); i++ )
        {
                printf( "Index: %.2d / Original: %.6f / Predicted: %.6f\n", i, original[ i ], predicted[ i ] );
                double delta = predicted[ i ] - original[ i ];
                error += delta * delta;
        }
        printf( "Burg Approximation Error: %f\n", error );

        return 0;
}
```