Prediction

Sébastien Boisgérault, Mines ParisTech

1 June, 2015

Contents

Prediction Principles

Prediction relies on the signal past and current values to estimate its future values. Such process relies on a given class of models, supposed to rule the behavior of the signal whose parameters shall be identified. This step being achieved, we may compute the **prediction error** or **residual**, the difference between the actual signal values and the predicted values. % In the context of data compression, and if the model used for prediction is accurate, the prediction

error has a much smaller range than the original values and therefore may be coded more efficiently.

Polynomial Prediction

Polynomial prediction is one of the simplest fixed-parameter prediction schemes. Given m sample values x_0, x_1, $\cdots$, x_{m-1}, we identify the unique polynomial P of order at most $m-1$ such that

$$\forall\, n \in \{0, 1, \cdots, m-1\},\ P(n) = x_n$$

and with it, provide a prediction $\hat{x}_m$ for the value x_m:

$$\hat{x}_m = P(m)$$

The polynomial P, given by

$$P(n) = \sum_{n=0}^{m-1} a_n j^n$$

is determined by the matrix equality

$$\begin{bmatrix} 1 & 0^1 & 0^2 & \dots & 0^{n-1} \\ 1 & 1^1 & 1^2 & \dots & 1^{n-1} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ 1 & (n-1)^1 & (n-1)^2 & \dots & n-1^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix}$$

The matrix on the left-hand side is an invertible Vandermonde matrix, therefore the polynomial coefficients may be obtained from the signal values x_0, x_1, $\cdots$, x_{n-1} and $\hat{x}_n$ can be computed.

The prediction error $e_n = x_n - \hat{x}_n$ may be computed efficiently. Consider a signal x_n whose values are stored in the NumPy array `x`, and let's begin with a polynomial prediction of order 0 or **difference coding** ; our prediction model is that the signal is constant. We may compute at once all the prediction errors $e_n = x_n - x_{n-1}$ for the signal with

```
e = diff(x) = [x[1] - x[0], x[2] - x[1], ...]
```

However, we end up with a vector with only `len(x) - 1` values : e_0 is undefined and `e[0]` would be e_1; we would have no information about the first value of the signal `x[0]` in `e`. We therefore add $x[0]$ as the first value of `e`. This is the same as taking into account a supposedly zero value `x[-1]` of the signal, and may by adding `0` to the beginning of x before applying the difference operator:

```
e = diff(r_[0, x])
```

Reconstruction of `x` from the residual `e` can be done by computing the cumulative sum $x_n = \sum_{i=0}^{n} e_n$:

```
x = cumsum(e)
```

What about first-order polynomial prediction then ? The formula for $\hat{x}_n$ is $\hat{x}_n = x_{n-1} + (x_{n-1} - x_{n-2}$ and the corresponding residual is

$$e_n = x_n - \hat{x_n} = x_n - 2x_{n-1} + x_{n-2} = (x_n - x_{n-1}) - (x_{n-1} - x_{n-2}).$$

This residual may therefore by computed as :

```
e_0 = diff(r_[0, x])
e = diff(r_[0, e_0])
```

and reconstruction is given as

```
e_0 = cumsum(e)
x = cumsum(e_0)
```

This scheme may be generalized to a polynomial prediction of arbitrary order.

Optimal Linear Prediction

Consider the following problem: given a sequence $\{x_n\}$, get the best linear approximation $\hat{x}_n$ of x_n as a linear combinations of the m previous samples:

$$\hat{x}_n = a_1 x_{n-1} + \cdots + a_m x_{n-m}.$$

Let's be more precise: if the values x_0, x_1, $\cdots$, x_{n-1} are available, we can predict $n-m$ values and therefore measure the prediction error by the quadratic criterion:

$$j(a) = \sum_{i=m}^{n-1} (x_i - a_1 x_{i-1} + \cdots + a_m x_{i-m})^2$$

The process that produces the estimates $\hat{x}_n$ is known as a **Wiener(-Hopf) filter**. % The vectors $a = (a_1, \cdots, a_m)$ that minimize the quadratic error are therefore solution of

$$a = \text{argmin}_x \|e\|^2, \quad \text{with} \quad e = Ax - b$$

where

$$A = \begin{bmatrix} x_{m-1} & x_{m-2} & \cdots & x_0 \\ x_m & x_{m-1} & \cdots & x_1 \\ \vdots & \vdots & \vdots & \vdots \\ x_{n-2} & x_{n-3} & \cdots & x_{n-m-1} \end{bmatrix} \quad \text{and} \quad b = \begin{bmatrix} x_m \\ x_{m+1} \\ \vdots \\ x_{n-1} \end{bmatrix}$$

The analysis of this problem shows that there is a unique solution $a = x$ if A is into that is if the square matrix $A^t A$ is full-rank (m) and the solution is

$$a = [A^t A]^{-1} A^t b$$

Indeed, the function $j(x) = \|Ax - b\|^2$ to minimize is quadratic in x: $j : x \mapsto 1/2x^tQx + Lx + c$. The Taylor decomposition at the point a yields $j(x) = j(a)+\nabla j(a)^t(x-a)+1/2(x-a)^t\nabla^2 j(a)(x-a)$. Any a such that $\nabla j(a) = 0$ (here, with the full rank assumption, there is a unique solution) is a global minimum. A geometrical analysis would also have worked: a solution a to the minimum problem has to be such that for any x, the error vector $e = b - Aa$ and Ax are orthogonal ; this also yields the condition (). The same geometrical anaylsis – or a direct computation – yields the error measure as by the Pythagorean Theorem, we have $\|b\|^2 = \|Aa\|^2 + \|Aa - b\|^2$

$$\|e\|^2 = \|b\|^2 - \|Aa\|^2$$

The full-rank assumption is not a problem in pratice: it just means that the signal data is rich enough to discriminate a unique optimal candidate x. If that's not the case, instead of $[A^tA]^{-1}A^t$ we could use the pseudo-inverse of $A^\sharp$ of A, defined as

$$A^\sharp = \lim_{\epsilon \to 0}[A^tA + \epsilon I]^{-1}A^t$$

such that $a = A^\sharp b$ provides among the solutions x of the minimisation problem the one with the smallest norm.

Instead of implementing ourself a solution of the minimization problem, we provide a reference implementation of the linear prediction problem that uses the NumPy function `linalg.lstsq` that solves this least-square (quadratic) minimization problem:

```
def lp(signal, m):
    "Wiener predictor coefficients"
    signal = ravel(signal)
    n = len(signal)

   A = array([signal[m - arange(1, m + 1) + i] for i in range(n-m)])
    b = signal[m:n]
    a = linalg.lstsq(A, b)[0]

    return a
```

Estimation of the parameter a as a solution of (+)
is called the **covariance method**. We preseent now a variant of this process, called **autocorrelation method**, that is amenable to faster implementations and also has more pleasant properties, such as the stability of the inverse of error filters (see section (**??**)).

Consider the following change: add m zeros at the start of $\{x_n\}$, add m zeros at the end, then apply the autocorrelation method. What we are trying to achieve is to predict ALL the values of x_n (even when we don't have all prior values) and conversely, for symmetry reasons that will be clearer in a moment, predict the trailing zeros from significant data as long as there is on usable sample.

The implementation of a linear predictor solver that support both methods is simple:

```
def lp(signal, m, zero_padding=False):
    "Wiener predictor coefficients"
    signal = ravel(signal)
   if zero_padding: # select autocorrelation method instead of covariance
        signal = r_[zeros(m), signal, zeros(m)]
    n = len(signal)

   A = array([signal[m - arange(1, m + 1) + i] for i in range(n-m)])
    b = signal[m:n]
    a = linalg.lstsq(A, b)[0]

    return a
```

Note that in the covariance methods, the new A and b are:

$$A = \begin{bmatrix} 0 & 0 & \dots & 0 & 0 \\ x_0 & 0 & \dots & 0 & 0 \\ x_1 & x_0 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ x_{n-2} & \dots & \dots & \dots & x_{n-m-1} \\ x_{n-1} & \dots & \dots & \dots & x_{n-m} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \dots & \dots & x_{n-1} & x_{n-2} \\ 0 & \dots & \dots & 0 & x_{n-1} \end{bmatrix} \quad \text{and} \quad b = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}$$

Set $x_i = 0$ if $i < 0$ or $i \geq n$ and

$$c_j = \sum_{i=-\infty}^{+\infty} x_i x_{i-j}$$

We now have

$$C(m) = A^t A = \begin{bmatrix} c_0 & c_1 & c_2 & \dots & c_{m-1} \\ c_1 & c_0 & c_1 & \dots & c_{m-2} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ c_{m-1} & c_{m-2} & \dots & c_1 & c_0 \end{bmatrix} \quad \text{and} \quad A^t b = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix}$$

The correlation matrix $C(m) = A^t A$ is is now symmetric but also **Toeplitz** (or **diagonal-constant**) and therefore efficient algorithms to solve the least-square problem exist. Note also that we could add MORE zeros before or after the data and that it wouldn't change a thing: A and b change but $A^t A$ and $A^t b$ are the same. In this context, the set of scalare equations

$$[A^t A]a = A^t b$$

are called **Wiener-Hopf** equatons, **Yule-Walker** equations or **normal equations**.

So the autocorrelation method naturally fits into the "infinite signals" point of view, strongly related to the convolution operator (see section **??**). To be more precise, consider the (causal) signal $\{x_n\}$ defined for ALL n (by setting 0 when not defined) and consider the signal $\{a_n\}$ defined in the same way (in particular, $a_0 = 0$). Then, the (possibly) non-zero coeffs of $\{a_n\} * \{x_n\}$ and $\{x_n\}$ correspond to the following vectors:

$$\{a_n\} * \{x_n\} \to Aa \text{ and } \{x_n\} \to b$$

so that the minimisation problem we are trying to solve really is:

$$\{a_n\} = \text{argmin} \, \|\{x_n\} - \{h_n\} * \{x_n\}\|^2$$

among all strictly causal filters h_n with $h_i = 0$ if $i > m$. Or if we introduce the prediction error filter $b_0 = 1$ and $b_n = -a_n$,

$$\{r_n\} = \text{argmin} \, \|\{h_n\} * \{x_n\}\|^2$$

among causal filters with $h_0 = 1$ and length less or equal to $m + 1$. This is a causal deconvolution problem.

Note however, the completion of the signal by 0's even if the result is questionable: if the real signal has values outside the window, they are probably not 0. It does not matter much when the length of the window is big with respect to the prediction order, but otherwise a the covariance method is probably more accurate.

Additional Properties of the Autocorrelation Method

Let $r_n = (1, -a1, \cdots, -a_m)$ be the coeffs of the prediction error filter. We are going to prove that:

$$\begin{bmatrix} c_0 & c_1 & c_2 & \dots & c_m \\ c_1 & c_0 & c_1 & \dots & c_{m-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ c_m & c_{m-1} & \dots & c_1 & c_0 \end{bmatrix} \begin{bmatrix} 1 \\ -a_1 \\ \vdots \\ -a_{m-1} \\ -a_m \end{bmatrix} = \begin{bmatrix} \|\{e_n\}\|^2 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}$$

By the way, that gives us a new way to get the a_n: get $C(m+1)^{-1}[1, 0, \cdots, 0]^t$ and normalize the result w.r.t. the first coefficient (whose meaning is interesting: the energy of the residual !). In the sequel, we denote $\sigma_r = \|\{e_n\}\|$.

All the 0 of the equations are a direct consequence of $AA^t a = A^t b$. The first coeff is equal to $c_0 - [c_1, \cdots, c_m] \cdot a = c_0 - (A^t b) \cdot a = c_0 - b \cdot (Aa) = \|\{x_n\}\|^2 - \{x_n\} \cdot (\{a_n\} * \{x_n\}) = \{x_n\} \cdot \{e_n\}$. But by the orthogonality condition, this is equal to $\{e_n\} \cdot \{e_n\} = \sigma_r^2$.

Linear Prediction of Unlimited Order – White Noise

Let $x_0, \cdots, x_{n-1}$ be a finite sequence of real values. We may extend the definition of x_i for arbitrary values of the index by setting $x_i = 0$ if i does not belong to the original index range. Now we may try to solve the linear prediction problem of unlimited order by minimizing over all *infinite* sequences of prediction coefficients a_i the quadratic sum of the prediction error e_i:

$$\sum_{i=-\infty}^{+\infty} e_i^2 \text{ where } e_i = x_i - \sum_{j=1}^{+\infty} a_i x_{i-j}$$

Any solution to this problem satisfies

$$\forall\, j > 0,\ \sum_{i=0}^{+\infty} e_i e_{i-j} = 0$$

The prediction error of the unlimited order linear prediction problem is not correlated at all – it is a **white noise.**

Proof. Let $(x * y)_i = \sum_{j=-\infty}^{+\infty} x_j y_{i-j}$, $\langle x, y\rangle = \sum_{i=-\infty}^{+\infty} x_i y_i$ and $\|x\| = \sqrt{\langle x, x\rangle}$. We denote by $L^2(\mathbb{Z})$ the set of infinite sequences x such that $\|x\| < +\infty$ and if $I \subset \mathbb{Z}$, by $L^2(I)$ the set of sequences x in $L^2(\mathbb{Z})$ such that $x_i = 0$ if $i \notin I$. Our minimization problem may be formalized as

$$\min_{a \in A} \|x - a * x\|^2 \text{ with } A = L^2(\mathbb{N}^*)$$

Let $e = x - a * x$ be the prediction error ; any solution a is a solution of () satisfies

$$\forall\, \delta \in L^2(\mathbb{N}^*),\ \langle \delta * x, e\rangle = 0$$

Let $\bar{x}$ be the infinite sequence such that $\bar{x}_i = x_{-i}$. We have $\langle \delta * x, e\rangle = \langle \delta, \bar{x} * e\rangle$ and $(\bar{x} * e)_j = \sum_{i=-\infty}^{+\infty} x_{i-j} e_i$. Therefore $\forall\, j > 0,\ \sum_{i=-\infty}^{+\infty} x_{i-j} e_i = 0$. As any e_i is a linear combination of the previous values of x, this equality yields

$$\forall\, j > 0,\ \sum_{i=0}^{+\infty} e_i e_{i-j} = 0$$

■

Finite Impulse Response (FIR) Filters

The Wiener-Hopf prediction that produces the sequence of estimates $\hat{x}_n$ from the x_n or error filter that outputs $e_n = x_n - \hat{x}_n$ are special cases of **finite impulse response (FIR) filters** : they associate to an input sequence u_n an output sequence y_n related by:

$$y_n = a_0 u_n + a_1 u_{n-1} + \cdots + a_{N-1} u_{n-N+1}$$

A core, real-time implementation for such system is given by:

```
class FIR(Filter):
    def __call__(self, input):
        if shape(input):
            inputs = ravel(input)
            return array([self(input) for input in inputs])
        else:
         output = self._a[0] * input + dot(self._a[1:], self.state)
            if len(self.state):
                self.state = r_[input, self.state[:-1]]
            return output
```

where some features, such as the initialization and changes of `a`, the management of the filter state, common between finite impulse response filters and auto-regressive filters (see section **??**) are implemented in the base class `Filter`. We talk about a *real-time* implementation of a FIR because instances of `FIR` produce the value y_n as soon as u_n is available. To do this, they need to store a state that contains at the time n the $N-1$ past values u_{n-1}, $\cdots$, u_{n-N+1} of the input.

Consider as an example the 4-point **moving average** filter:

$$y_n = \frac{1}{4}(u_n + u_{n-1} + u_{n-2} + u_{n-3})$$

Such a filter may be defined and used by the following code:

```
>>> ma = FIR([0.25, 0.25, 0.25, 0.25])
>>> ma.state
array([ 0.,  0.,  0.])
>>> ma(1.0)
0.25
>>> ma(2.0)
0.75
>>> ma(3.0)
1.5
>>> ma(4.0)
2.5
>>> ma([5.0, 6.0, 7.0, 8.0, 9.0, 10.0])
array([ 3.5,  4.5,  5.5,  6.5,  7.5,  8.5])
>>> ma.state
array([ 10.,   9.,   8.])
```

Once the filter `ma` is initialized (by default with a zero state), every call to `ma` shall give one or several new input values and as many output values are produced.

Note that if we start with a zero state and input a single non-zero value before sending a sequence of zeros, the filter will output a finite number of (possibly) non-zero and will then output only zeros: this is actually a defining property of finite impulse response filters.

```
>>> ma.state = [0.0, 0.0, 0.0]
>>> ma([1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0])
array([ 0.25,  0.25,  0.25,  0.25,  0.  ,  0.  ,  0.  ,  0.  ,  0.  ])
```

In the context of linear prediction, here is how the prediction coefficients `a` produced by `lp` may be used to build the predictor filters and error filters.

```
>>> a = lp(x, ...)
>>> predictor = FIR(r_[0.0, a])
>>> error = FIR(r_[1.0, -a])
```

Auto-Regressive (AR) Filters

Autoregressive (AR) filters are – at least formally – inverses of FIR filters. Consider the equation of an FIR error filter whose input is x_n and output e_n

$$e_n = 1.0 \cdot x_n - a_1 x_{n-1} - \cdots - a_m x_{n-m}$$

If this equation holds for every value of n, the inverse system that has e_n as an input and x_n as an output is ruled by:

$$x_n = a_1 x_{n-1} + \cdots + a_m x_{n-m} + e_n$$

Therefore we consider the class of autoregressive systems with inputs u_n and outputs y_n ruled by

$$y_n = a_1 y_{n-1} + \cdots + a_N y_{n-N} + u_n$$

A core implementation is given as

```
class AR(Filter):
    def __call__(self, input):
        if shape(input):
            inputs = ravel(input)
            return array([self(input) for input in inputs])
        else:
            output = dot(self.a, self.state) + input
            self.state[-1] = output
            self.state = roll(self.state, 1)
            return output
```

The state of such an AR instance is the sequence of N previous values of y_n. The usage of the class `AR` is similar to `FIR`. For example, the filter:

$$y_n = 0.5 \cdot y_{n-1} + u_n$$

may be defined and used by

```
>>> ar = AR([0.5])
>>> ar.state = [1.0]
>>> ar(0.0)
0.5
>>> ar(0.0)
0.25
>>> ar(0.0)
0.125
>>> ar(0.0)
0.0625
>>> ar([1.0, 1.0, 1.0, 1.0])
array([ 1.03125   ,  1.515625  ,  1.7578125 ,  1.87890625])
>>> ar.state
array([ 1.87890625])
```

Transfer Function, Stability and Frequency Response

Transfer Function

The **transfer function** of a (linear, time-invariant, single-input single output) system is a (partial) function $H : \mathbb{C} \to \mathbb{C}$ defined in the following way: given $z \in \mathbb{C}$ and a complex-valued input signal $u_n = uz^n$ the corresponding output having the structure $y_n = yz^n$, if it exists, satisfies:

$$y = H(z)u$$

For example, the FIR filter defined by the equation () has the transfer function

$$H(z) = a_0 + a_1 z^{-1} + \cdots + a_{N-1} z^{-N+1}$$

and the AR filter defined by the equation () has the transfer function

$$H(z) = \frac{1}{1 - a_1 z^{-1} - \cdots - a_N z^{-N}}$$

Stability

A filter is **(input-output) stable** if all bounded input signals result in bounded outputs. Stability of filters whose transfer function is rational – such as FIR and AR filters – is conditioned by the location of their **poles**, the roots of their transfer functions. Precisely, such a filter is stable if and only if all its poles have a negative real part.

The classes `FIR` and `AR` have a method that return their poles ; its implementation is based on the `numpy.lib.polynomial roots` function that computes the roots of a polynomial.

For FIR filters, the situation is simple: as

$$H(z) = a_0 + a_1 z^{-1} + \cdots + a_{N-1} z^{-N+1} = \frac{a_0 z^{N-1} + a_1 z^{N-2} + \cdots + a_{N-1}}{z^{N-1}},$$

all N poles are 0 and therefore all FIR filters are stables.

```
class FIR(Filter):
    ....
    def poles(self):
        return zeros(len(self.a))
```

For AR filters,

$$H(z) = \frac{1}{1 - a_1 z^{-1} - \cdots - a_N z^{-N}} = \frac{z^N}{z^N - a_1 z^{N-1} - \cdots - a_N}$$

and therefore the poles are the solution of the polynomial $P(z) = z^N - a_1 z^{N-1} - \cdots - a_N$.

```
class AR(Filter):
    ...
    def poles(self):
        return roots(r_[1.0, -self.a])
```

As an example, consider the two auto-regressive filters ruled by:

$$y_n = 0.5 \cdot y_{n-1} - 0.5 \cdot y_{n-2} + u_n \text{ and } y_n = y_{n-1} + y_{n-2} + y_{n-3} + y_{n-4} + u_n$$

The first one is stable but the second one is unstable:

```
>>> ar = AR([0.5, -0.5])
>>> ar.poles()
array([ 0.25+0.66143783j,  0.25-0.66143783j])
>>> max(abs(pole) for pole in ar.poles())
0.70710678118654757
>>> ar = AR([1.0])
>>> ar = AR([1.0, 1.0, 1.0, 1.0])
>>> ar.poles()
array([ 1.92756198+0.j        , -0.77480411+0.j        ,
       -0.07637893+0.81470365j, -0.07637893-0.81470365j])
>>> max(abs(pole) for pole in ar.poles())
1.9275619754829254
```

As a matter of fact, we will deal in the next sections with AR filters that are inverses of FIR prediction error filters provided by linear prediction. Such filters are always stable when the autocorrelation method is used but may be unstable with the covariance method.

Frequency Response

When a filter is stable, it makes sense to ask what output corresponds to a cosine input with frequency f, amplitude A and phase ϕ. If the input sequence is scheduled to produce a new value every Δt seconds, we have

$$u_n = A \cos 2\pi f n \Delta t + \phi$$

and therefore

$$u_n = A/2 \cdot e^{i\phi}(e^{i2\pi f \Delta t})^n + A/2 e^{-i\phi} \cdot (e^{-i2\pi f \Delta t})^n.$$

By the definition of the transfer function, we have the corresponding output y_n:

$$\begin{aligned} y_n &= A/2 \cdot e^{i\phi} H(2\pi f \Delta t)(e^{i2\pi f \Delta t})^n + A/2 \cdot e^{-i\phi} H(-2\pi f \Delta t)(e^{-i2\pi f \Delta t})^n \\ &= \mathbb{R}e\left[H(2\pi f \Delta t) A e^{i\phi} e^{i2\pi f \Delta t n + \phi}\right] \end{aligned}$$

So if we consider the polar decomposition

$$H(2\pi f \Delta t) A e^{i\phi} = A' e^{i\phi'}$$

then the cosine output of the filter is

$$y_n = A' \cos 2\pi f n \Delta t + \phi'$$

The function

$$f \mapsto H(2\pi f \Delta t)$$

that relates input and ouput amplitude and phase at the frequency f is called the filter {frequency response}. We often consider separately

$$|H(2\pi f \Delta t)| \text{ and } \angle H(2\pi f \Delta t),$$

the frequency response **gain** and **phase**.

The implementation of transfer functions for FIR and AR filters relies on the computation of signal spectrum or Fourier transform, provided by the function `F` of the `spectrum` module, see section **??**.

```
from spectrum import F

class FIR(Filter):
    ...
    def __F__(self, **kwargs):
        dt = kwargs.get("dt") or 1.0
        return F(self.a / dt, dt=dt)

class AR(Filter):
```

```
    ...
    def __F__(self, **kwargs):
        dt = kwargs.get("dt") or 1.0
        FIR_spectrum = F(FIR(a=r_[1.0, -self.a]), dt=dt)
        def AR_spectrum(f):
            return 1.0 / FIR_spectrum(f)
        return AR_spectrum
```

The function `F` is generally used to get the frequential representation of and object, signal or filter, or something else. Apart from signals, for which we directly compute the spectrum, the objects are supposed to know what their spectral representation is and encode this information in the special method `__F__` ; for filters, we return the frequency response. Those methods being defined for FIR and AR filters, we may use them like that:

```
>>> ma = FIR([0.5, 0.5])
>>> tf = F(ma, dt=1.0)
>>> tf([0.0, 0.1, 0.2, 0.3, 0.4, 0.5])
array([ 1.0000000 +0.00000000e+00j,  0.9045085 -2.93892626e-01j,
        0.6545085 -4.75528258e-01j,  0.3454915 -4.75528258e-01j,
        0.0954915 -2.93892626e-01j,  0.0000000 -6.12303177e-17j])
```

Voice Analysis and Synthesis

The TIMIT corpus

The TIMIT corpus is a collection of read speech data that includes for each utterance 16-bit 16 kHz waveforms as well as time-aligned orthographic, phonetic and word transcriptions. It was designed – as a joint effort among the Massachusetts Institute of Technology (MIT), SRI International (SRI) and Texas Instruments, Inc. (TI) – for acoustic-phonetic studies and for the development and evaluation of automatic speech recognition systems.

The Python library NLTK – for Natural Language Toolkit – is an open source collection of modules that provides linguistic data and documentation for research and development in natural language processing and text analytics (`http://www.nltk.org/`). As a part of the distribution, is a small sample of the TIMIT corpus is made available.

The samples from TIMIT are designated by ids whose list is given by the `utteranceids` method:

```
>>> import nltk
>>> timit = nltk.corpus.timit
>>> timit.utteranceids()
['dr1-fvmh0/sa1', 'dr1-fvmh0/sa2', 'dr1-fvmh0/si1466', 'dr1-fvmh0/si2096',
 ...
```

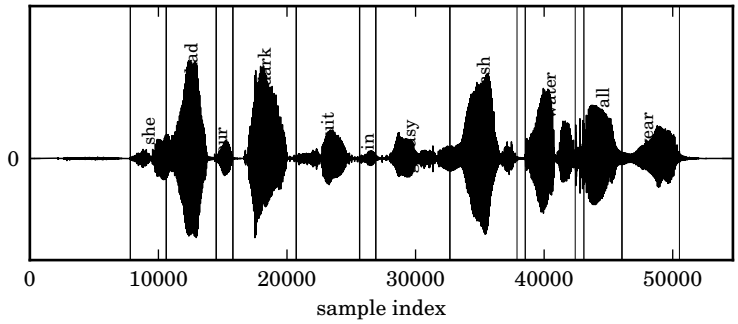

Figure 1: Waveform of the TIMIT utterance `'dr1-fvmh0/sa1'` and display of its segmentation into words.

```
 'dr8-mbcg0/sx237', 'dr8-mbcg0/sx327', 'dr8-mbcg0/sx417', 'dr8-mbcg0/sx57']
>>> uid = timit.utteranceids()[0]
'dr1-fvmh0/sa1'
```

The corpus provides a detailled decomposition of the utterances in words as well as **phones**[1] – speech segments that have distinct properties. Those decompositions are timed, the numbers being sample indices.

```
>>> timit.words(uid)
['she', 'had', 'your', 'dark', 'suit', 'in', 'greasy', 'wash', 'water', 'all', 'year']
>>> timit.word_times(uid)
[('she', 7812, 10610), ('had', 10610, 14496), ('your', 14496, 15791),
 ('dark', 15791, 20720), ('suit', 20720, 25647), ('in', 25647, 26906),
 ('greasy', 26906, 32668), ('wash', 32668, 37890), ('water', 38531, 42417),
 ('all', 43091, 46052), ('year', 46052, 50522)]
>>> timit.transcription_dict()["she"]
['sh', 'iy1']
>>> timit.phones(uid)
['h#', 'sh', 'iy', 'hv', 'ae', 'dcl', 'y', 'ix', 'dcl', 'd', 'aa', 'kcl', 's',
 'ux', 'tcl', 'en', 'gcl', 'g', 'r', 'iy', 's', 'iy', 'w', 'aa', 'sh', 'epi',
 'w', 'aa', 'dx', 'ax', 'q', 'ao', 'l', 'y', 'ih', 'ax', 'h#']
>>> timit.phone_times(uid)
[('h#', 0, 7812), ('sh', 7812, 9507), ('iy', 9507, 10610), ('hv', 10610, 11697),
 ...
 ('ih', 47848, 49561), ('ax', 49561, 50522), ('h#', 50522, 54682)]
```

The `audiodata` method, combined with the `bitstream` module, provide the waveform as a single-dimensional NumPy array `data`:

```
>>> str_data = timit.audiodata(uid)
```

[1] not to be confused with **phonemes**, set of phones that are cognitively equivalent (<http://en.wikipedia.org/wiki/Phoneme>).

```
>>> data = BitStream(str_data).read(int16, inf).newbyteorder()
```

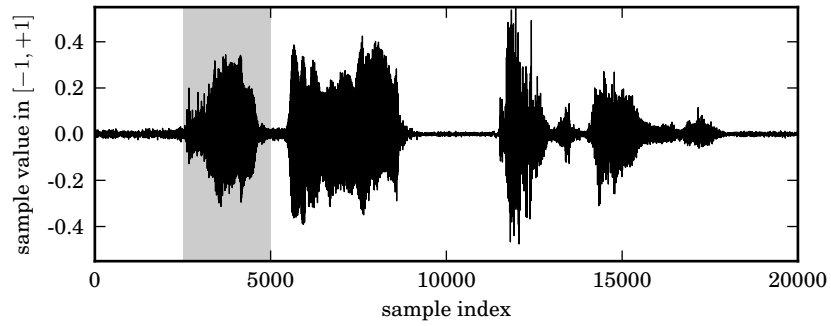

Figure 2:

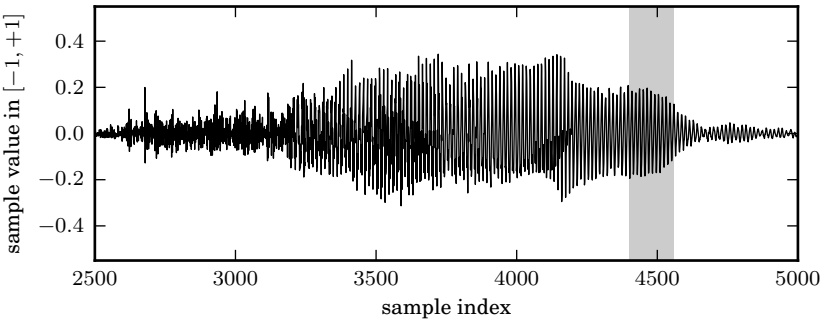

Figure 3:

Voice Analysis and Compression

The knowledge that the audio data that we are willing to compress is a voice signal can go a long way in the reduction of bit rate. Consider for example the G.711 PCM speech codec: defined in 1972, it is based on a 8 kHz sampling time and a quite generic method of non-linear quantization (8-bit μ-law or A-law). It achieves a data rate of 64 kb/s. A more specific technology developed in the early 90's, and based on linear prediction, the full-rate GSM, has a 13 kbps bit rate. More recent efforts in this direction have achieved a quality similar to the G.711 codec at 6.4 kbps, or with a lesser quality go as low 2.4 kbps (see [**?**]).

In the sequel, we'll assume that the data we consider is sampled at 8 khz ; this is a standard assumption in fixed telephony that takes into account the fact that most voice audio content is in the 300-3400 Hz band. Applications that require

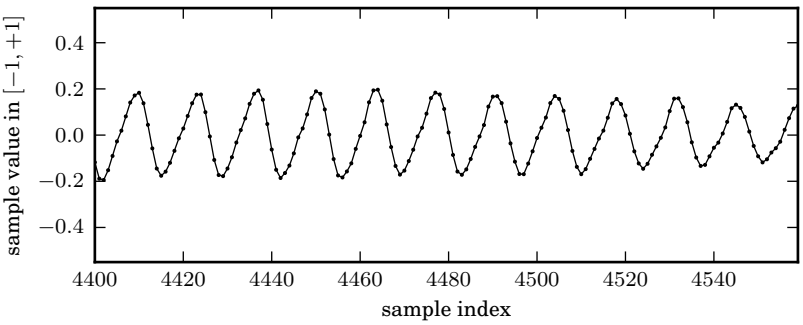

Figure 4: **Voice patterns.** A voice signal sampled at 8 kHz displays complex and non-stationary patterns on a scale of 2.5 s (top). When we zoom to a 300 ms scale (middle), and then further to a 20 ms scale (bottom), we see that locally, the signal appears to be almost periodic.

more accurate descriptions of voice data may use **wideband** and use a 16 kHz sampling instead for a higher accuracy – all the audio data in the TIMIT data base uses this sampling frequency for example.

Short-Term Prediction

Beyond the selection of an appropriate sampling rate, the key to achieve significant compression rate is to recognize that voice has a local – say on a 20 ms frame – stationary structure that can therefore be described by a small numbers of parameters. This property is clearly visible in the figure **??**.

The figure **??** displays two 20-ms voice fragments sampled at 8 kHz and the corresponding residuals after a prediction of order . The first one clearly has achived its goal: the residual appears to be left without structure and is a good approximation of a white noise. For those kind of data, the short-term prediction provides a simple production model: an AR synthesis filter whose input is a white noise. For the second type of signals, for which the residual is clearly not random, we need a more complex production model that complements the short-term prediction with a long term prediction (see sections **??** and {**??**}).

Spectral Analysis

The spectrum of a voice segment $x(t)$ may be estimated classically, with the formula

$$x(f) = \Delta t \sum_{t \in \mathbb{Z}\Delta t} x(t) \exp(-i2\pi f t),$$

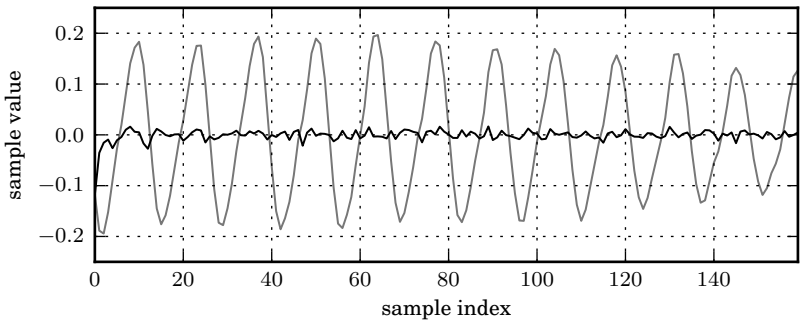

Figure 5:

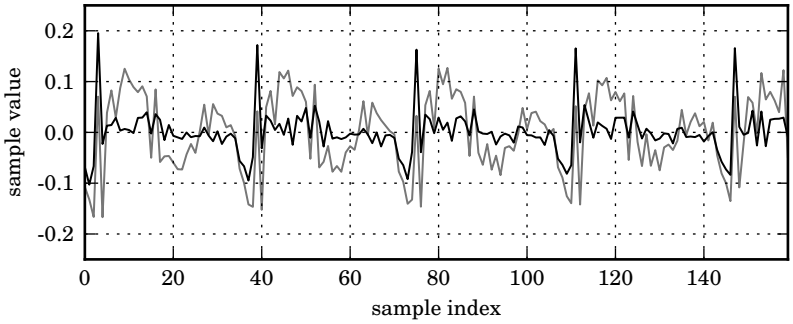

Figure 6: **short-term prediction error.** top: a frame of 160 samples within a 8 kHz signal (grey) and the corresponding prediction error (black) for a linear prediction of order 16 (covariance method). The residual show little remaining structure. bottom: the prediction error (black) of the voice signal (grey) still exhibits a periodic structure, made of regularly spaced spikes, characteristic of voiced segments.

but there is another way: if we have performed a successful predicton of the data that leads to a synthesis filter with frequency response

$$\frac{1}{1-A(f)},$$

the prediction error should be almost white and its spectrum $e(f)$ should be approximately constant. As the signal data $x(t)$ is related to $e(t)$ by

$$x(f) = \frac{1}{1-A(f)}e(f),$$

the frequency response of the synthesis filter provides a (parametric) estimate of the signal spectrum. Both kind of methods are illustrated in figure **??**.

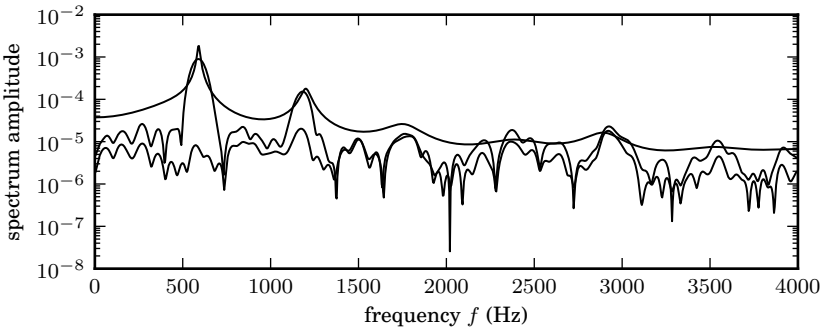

Figure 7: **Spectral View.** spectrum of the signal of the figure **??**, estimated by non-parametric (fft) method and by the frequency response of the synthesis filter. The spectrum of the prediction error is also displayed.

Models of the Vocal Tract

Continuous Modelling. % A simple model of vocal tract is the **horn**: a tube whose cross-sectional area A is a function of the position x in the tube. Let ϕ denote the air flow, positive by convention if the are travel towards the right, p the pressure, ρ the air density and K its bulk modulus.

Newton's second law of motion yields

$$\frac{d}{dt}\rho\phi = -\frac{dpA}{dx}.$$

We approximate this equation by:

$$\rho\frac{\partial\phi}{\partial t} = -A\frac{\partial p}{\partial x}$$

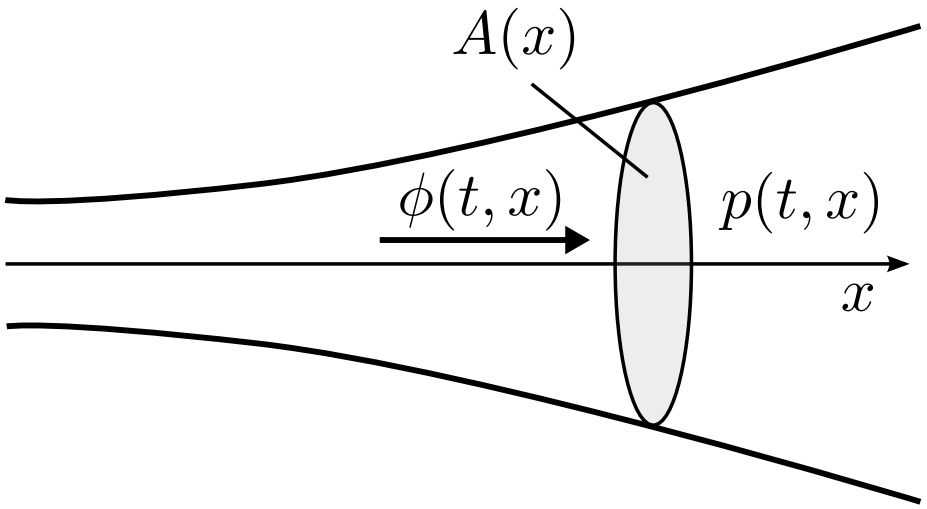

Figure 8: the vocal tract: horn model.

On the other hand, as the bulk modulus relates changes in the pressure p and in the volume by $ K dv + v dp = 0, $ we also have

$$K\frac{\partial \phi}{\partial x} = -A\frac{\partial p}{\partial t}$$

The combination of equations () and () yield **Webster's Equation**

$$\frac{1}{c^2}\frac{\partial^2 p}{\partial t^2} - \frac{1}{A}\frac{dA}{dx}\frac{\partial p}{\partial x} - \frac{\partial^2 p}{\partial x^2} = 0$$

where c, the wave velocity in the media is given by:

$$c = \sqrt{\frac{K}{\rho}}$$

Discrete Modelling. An common simplification of the horn model is to trade the continuous change in the cross-sectional area $A(x)$ for a tube made of a finite number of cylindrical sections of equal length L whose cross-sectional area A_k is a function of the section index k. Consider the pressure p_k in the section k as the superposition of right and left-travelling waves $ p_k(t,x) = p_kˆ+(x - ct) - p_kˆ-(x + ct). $

Stating that the pressure is continuous at the section boundary x leads to the system of equations

$$\begin{array}{lcll} p^+_{k+1}(x-ct) & = & (1-r^+_k)p^+_k(x-ct) & + \quad r^-_{k+1}p^-_{k+1}(x+ct) \\ p^-_k(x+ct) & = & (1-r^-_{k+1})p^-_{k+1}(x+ct) & + \quad r^+_k p^+_k(x-ct) \end{array}$$

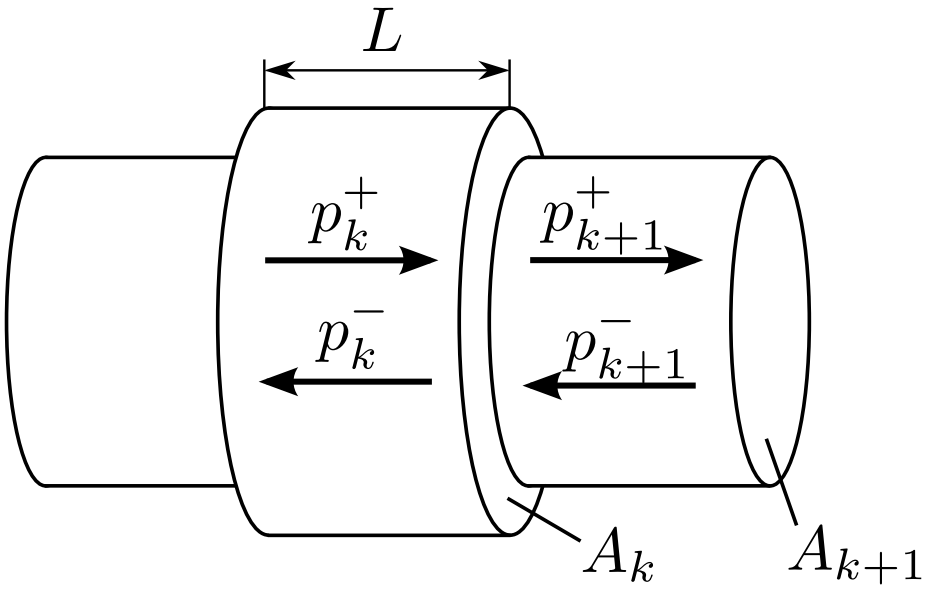

Figure 9: the vocal tract: discrete tube model.

where r_k^+ and r_{k+1}^- are **reflection coefficients**. The air flow

$$\phi_k(t,x) = \phi_k^+(x-ct) - \phi_k^-(x+ct)$$

is also continuous at the section boundary. A Fourier decomposition of the waves and the use of equation () show that it is related to the pressure by

$$\frac{p_k^\pm}{\phi_k^\pm} = \pm Z_k$$

where Z_k is the **impedance**, given in each section by

$$Z_k = \frac{c\rho}{A_k} = \frac{\sqrt{K\rho}}{A_k}$$

The continuity of the air flow at the position x provides for all time the equations

$$A_k p_k^+(x-ct) + A_k p_k^-(x+ct) = A_{k+1} p_{k+1}^+(x-ct) + A_{k+1} p_{k+1}^-(x+ct)$$

which, coupled with the system of equations () leads to

$$r_k^+ = -r_{k+1}^- = \frac{A_{k+1} - A_k}{A_{k+1} + A_k}.$$

Ladder and Lattice Filters. In a given tube section, the pressure waves travel unchanged at the speed c. Given that the tube section is of length L, the

time needed to go from one boundary of the section to the other is L/c. As a consequence, the transformation between the values of p^+ and p^- from the left of one section boundary to the left of the next section boundary on the right may be modelled as the junction depicted on the left of figure **??**.

Now, if want to follow what happens to the pressure wave p^+ travelling to the right, we may introduce a variable $\tilde{p}^+$ that compensates for the delay in the wave propagation as well as the attenuation (or amplification) at the sections boundary. We apply the same treatment to $\tilde{p}^-$ so that

$$\tilde{p}^{\pm}_{k+1}(t) = \frac{1}{1-r_k} p^{\pm}_{k+1}(t + L/c)$$

Straightforward computations show that the equations satisfied by the corresponding variables are described by the lattice junction depicted on the right of the figure **??**.

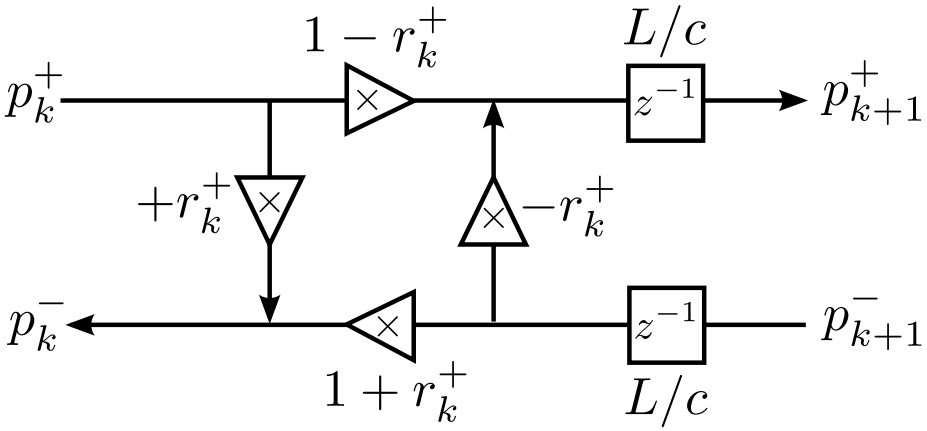

Figure 10:

Lattice Filters in Linear Predictive Coding. **When it comes to the implementation of synthesis filters that model the vocal tract, lattice filters – implemented as a serial connexions of lattice junctions – are often preferred to classic (register-based) implementations of the autoregressive filters. Their parameters – the reflection coefficients – are easy to interpret and also, when the synthesis filter is determined by the autocorrelation method, the Levison-Durbin **or** Schur** algorithm may be used to compute them directly instead of the linear regression coefficients a_i. Moreover, these algorithms are recursive and have
a $\mathcal{O}(m^2)$ complexity where m is the predicton order, better than the typical $\mathcal{O}(m^3)$ of the least-square resolution needed to compute the a_i.

Finally, lattice filters are stable as long as the reflexion coefficients are between -1 and 1. As a consequence, we can easily perform a quantization of these coefficients that will preserve the stability of the synthesis filter. A classic choice

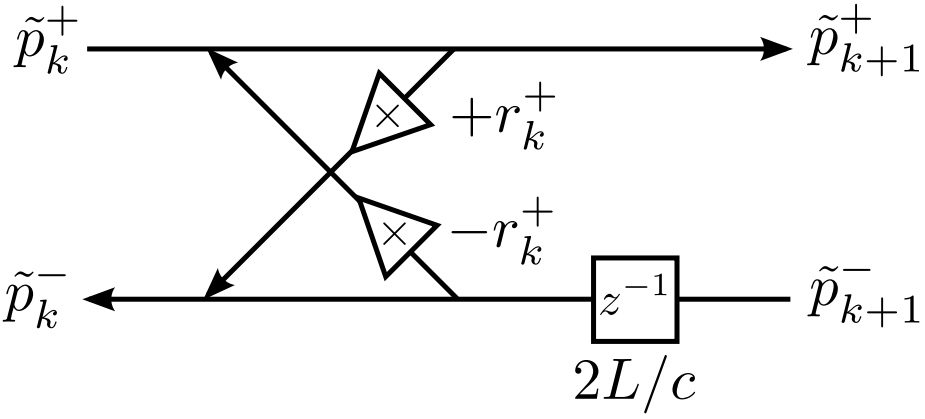

Figure 11: Kelly-Lochbaum junction: ladder form (left) and lattice form (right)

is the logarithmic quantization of the area-ratio A_{k+1}/A_k, that is, because of the equation (), the uniform quantization of

$$\mathrm{LAR}_k = \log \frac{1 + r_k^+}{1 - r_k^+}.$$

Pitch Analysis

The prediction error of the voice fragment displayed at the bottom of figure **??** still displays some structure : a white noise plus a quasi-periodic sequence of impulses. As the short-term prediction has inverted the vocal tract filter, what we are looking at is actually the sequence of **glottal pulses**. The duration between two pulses is the voice **pitch period**, ts inverse is the speech **fundamental frequency**. When this periodic structure is present after short-term predicton, the speech fragment is said to be **voiced** and when it's not, it is **unvoiced**.

The simplest kind voiced/unvoiced classifier is based on the **autocorrelation** of the short-term prediction error (see for example [**?**]). In a given data frame, we select a subframe, typically at the end, and compare it with all the subframes of equal size within the frame by computing the normalized scalar product between the two vectors. Values of (the modulus of) the correlation near 1 correspond to two subframes that are – up to a gain – almost equal.

```
def ACF(data, frame_length):
    frame = data[-frame_length:]
    frame = frame / norm(frame)
    past_length = len(data) - frame_length
    correl = zeros(past_length + 1)
    for offset, _ in enumerate(correl):
```

```
  past_frame = data[past_length-offset:past_length-offset+frame_length]
     past_frame = past_frame / norm(past_frame)
     correl[offset] = dot(past_frame, frame)
return correl
```

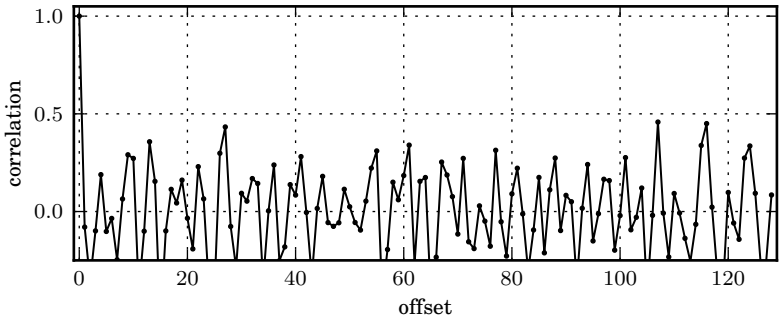

Figure 12:

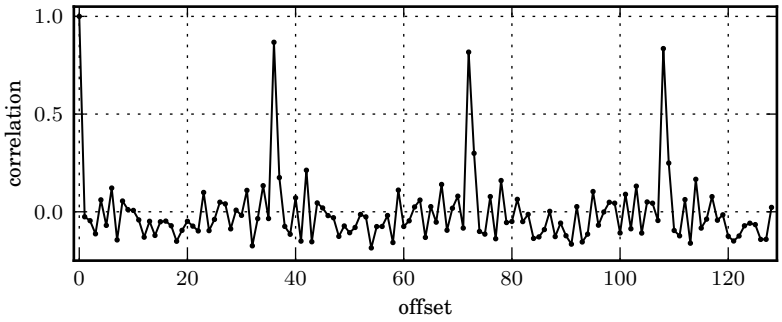

Figure 13: normalized autocorrelation of the prediction errors for the speech fragment of figure **??** with a reference window of 32 samples. The top graph corresponds to an unvoiced signal and the bottom one to a voiced signal with a pitch period of 36 samples.

These kind of method will therefore rely on the selection of autocorrelation threshold to distinguish between voiced and unvoiced signals and localization of autocorrelation maxima to estimate the pitch period. Care must be taken not to select a multiple of the pitch period instead.

Long-Term Prediction

Given a reference subframe y and a subframe x offsetted by the pitch period p we can compute the best linear approximation of y in terms of x, that is, the gain k, solution of

$$k = \text{argmin}_{\kappa} \|y - \kappa x\|^2.$$

It is given by

$$k = \frac{x^t y}{\|x\|^2}$$

Once again, what we have done is a prediction, but a long-term prediction, applied to the residual of the short-term prediction. If x_n denotes the error of the short-term prediction, the error e_n after the additional long-term prediction is given by

$$x_n = k x_{n-p} + e_n$$

This equation models an auto-regressive synthesis filter whose diagram is given in figure **??**

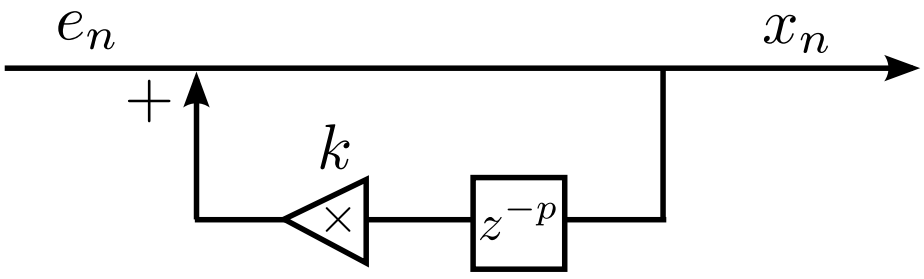

Figure 14: LTP synthesis filter

Linear Prediction Coding

The use of short-term and long-term linear prediction method may be used in several ways to compress voice information. The algorithms that follow this path are generally referred to as **Linear Predictive Coding (LPC)**. "Pure" LPC algorithms encode the prediction parameters and the residual power but do not keep any extra information on the prediction residual ; this approach is consistent with the belief that a good prediction produces a residual which is a white noise. The voice is reconstructed by the injection of a synthetic white noise into the synthesis filter.

Adaptative Predictive Coding (APC) is also called **Residual-Excited Linear Prediction (RELP)** : in order to have a reconstructed voice with

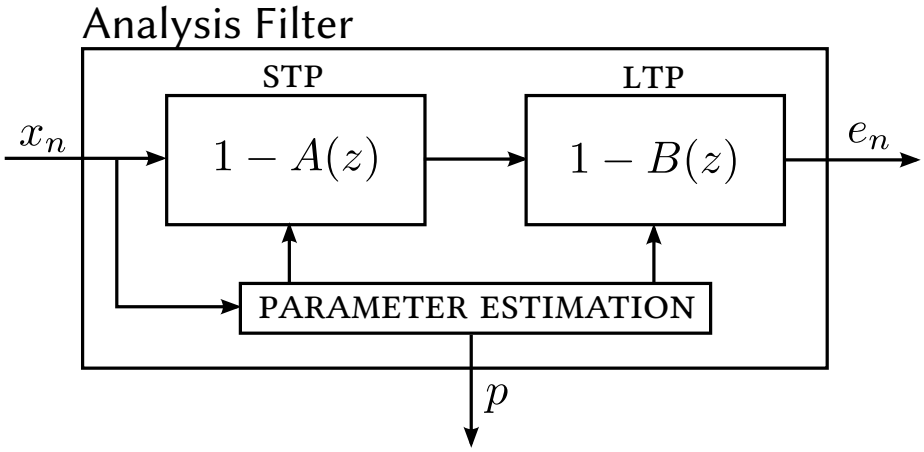

Figure 15:

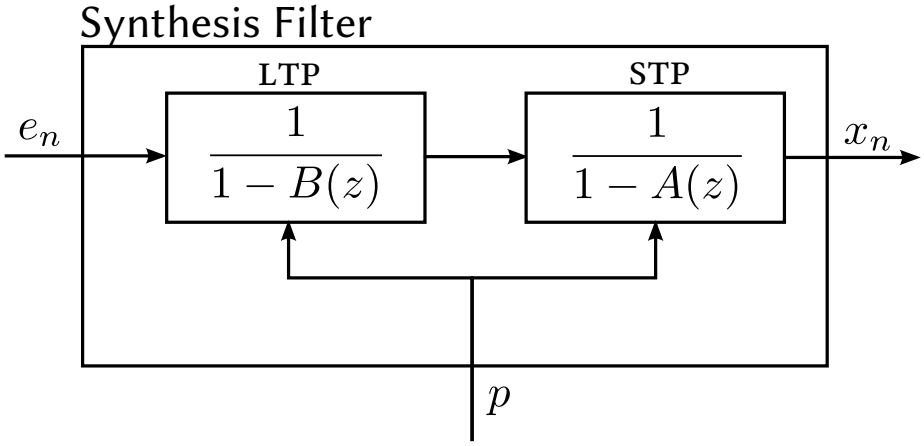

Figure 16: LPC analysis and synthesis filters diagram}

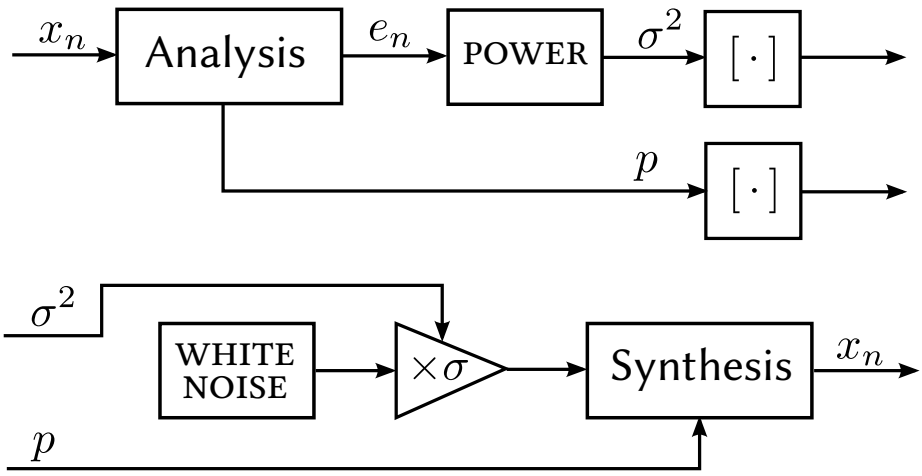

Figure 17: "Pure" LPC analysis and synthesis diagrams

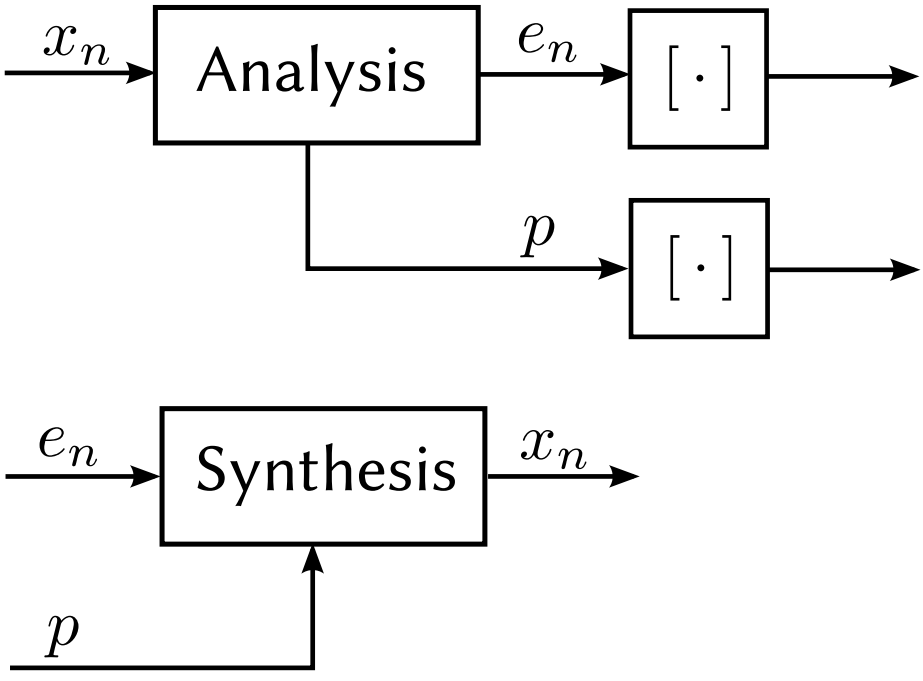

Figure 18: APC/REPL analysis and synthesis diagrams

a higher quality, the residual information is not discarded but quantized and transmitted along with the prediction parameters.

This kind of approach has a major drawback: the quantization typically aims at the minimization of the quantization error *of the residual*, a quantity that has little to do with the error induced on the voice itself. The **Code-Excited Linear Prediction (CELP)** approach solves that issue by discarding the residual entirely and by trying instead several excitation signals among a finite **codebook**, apply to them the synthesis filter, and look for the output that matches the more closely the voice data.

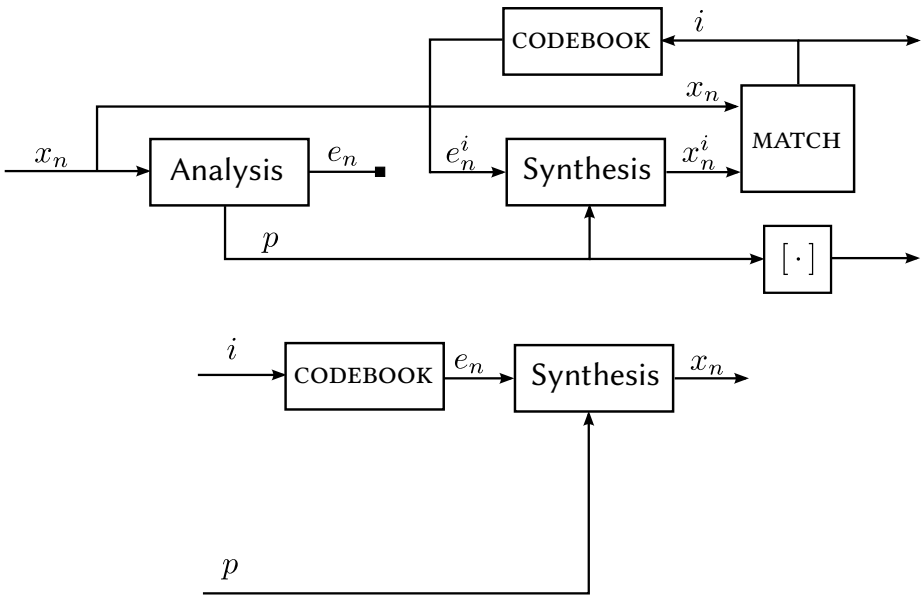

Figure 19: CELP analysis and synthesis diagrams