RTA Documentation
Using the *RTA* library to compute on-line sound descriptors

Jean-Philippe Lambert

9th May 2008 (revision 1.6)

Contents

List of Figures

1 Doxygen: Code self-contained documentation

The *RTA* library is documented within the code, using the *Doxygen* format. This is the reference documentation. To generate *HTML* and *LaTeX* output, use the command `doxygen Doxyfile` within the `rta` directory. The *HTML* documentation will then be accessible from `rta/doc/html/index.html`. To compile a *PDF* document, use the command `make` within the `rta/doc/latex` directory, which is generated by the previous command.

2 Generalities

The *RTA* library is frame-based, which means that for any vectors of samples, a set of descriptors can be computed without adding any delay. A noticeable exception to this is the *delta* (and *delta-delta*) computation, as it is based on more than on frame.

There are two variants of each function, one being `_stride` post-fixed. It allows to directly access interleaved data without copying them (like a stereophonic samples vector). However, using a big *stride* value may lead to a bad usage of the memory cache.

Any index starts at 0.

Some functions require an initialisation before any processing, in order to pre-calculate what will not depend on the incoming frame. Every allocation must be done before anything else, outside of the functions themselves except mentioned otherwise.

Some descriptors can be computed by several functions, and the results may slightly differ for several reasons: the functions does not rely on the same algorithms and the signal used for the computation may differ (due to windowing, filtering, etc.). The auto-correlation from *yin* and from the *LPC* are not the same and there is a lot of ways to get the energy: from *yin*, as the sum of the squares of the samples, from *LPC*, or as the first *MFCC*coefficient.

3 Library configuration

A file named `rta_configuration.h` must be present within your sources in order to use the *RTA* library. An empty file means that the defaults settings are used for the compilation.

Instead of using the `malloc`, `realloc` and `free` functions from the `stdlib.h`, one can respectively define `rta_malloc`, `rta_realloc` and `rta_free`.

The floating-point precision can be simple, double or long double, according to the definition of `RTA_REAL_TYPE` to respectively `RTA_FLOAT_TYPE`, `RTA_DOUBLE_TYPE` or `RTA_LONG_DOUBLE_TYPE`. The constants in `rta_float.h` are then redefined according to the proper type from `float.h`. The same applies to the functions in `rta_math.h` from `math.h`.

Note that the long double precision is not supported when using the Apple's

VecLib by setting `RTA_USE_VECLIB` to 1.

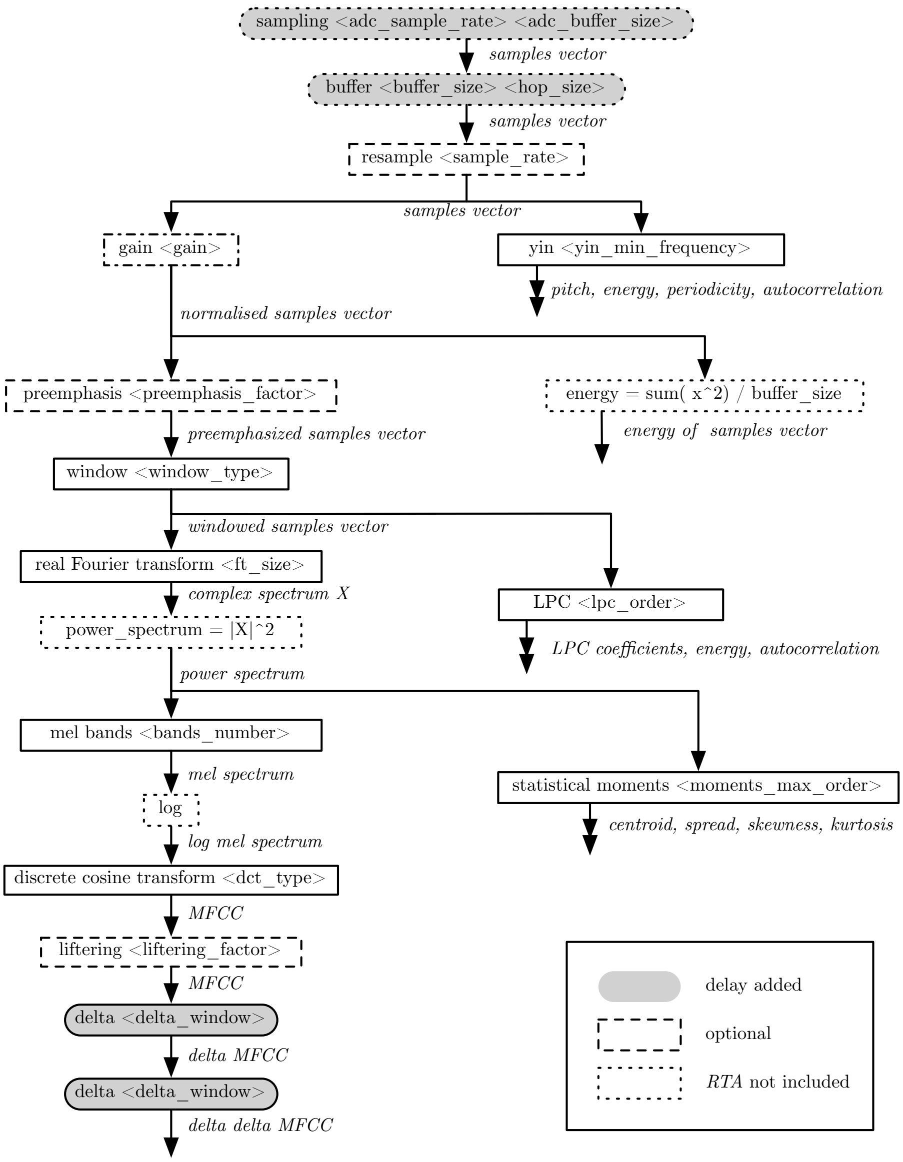

Figure 1: Sound descriptors data-flow

4 Descriptors based on a vector of samples

The vector of samples is characterised by its sample-rate and its size. Moreover, to use sizes larger than those provided by the sound card (e.g. for Fourier transforms), the hop-size gives the number of samples between two consecutive vectors that overlap if the hop-size is smaller than the vector-size.

4.1 Re-sampling

Down-sampling a signal is interesting to lower the further computations, especially for the computational-intensive algorithms, like *yin*. It can helps to concentrate on a range of the spectrum where the information is pertinent for the analysis: the *LPC* (which is linear, as its name suggests it) finds poles and zeros to fit the whole spectrum; to keep the results under 5 kHz, we simply use a sample-rate of 11 kHz. If the original signal sample-rate is 44 kHz, we can use the function `rta_downsample_int_mean` with a factor of 4. This function implies a low-pass filtering. The function `rta_downsample_int_remove` can produce aliasing if the original signal contains information above the resulting sample-rate. Note that the resulting samples vector-size is smaller, according to the given factor.

4.2 *Yin*

The *yin* algorithm computes the periodicity of a samples vector, finding its most probable *lag*. To get the period in Hz, on simply multiply this lag by the input vector sample-rate.

Note that the *yin* algorithm operates on a non-windowed samples vector. As it can be computational-intensive, a down-sampling of the incoming vector is often performed: to track a pitch below 1 kHz, one can use a sampling-rate of 11 kHz, or even 5.5 kHz, depending on the results quality request. One can then check the *absolute minimum* found, which gives the *periodicity* as long as the absolute minimum is positive:

$$periodicity = 1 - \sqrt{absolute\_minimum}$$

Before computing anything, a new *yin* structure must be allocated and filled with the `rta_yin_setup_new` function. It will be released by the function `rta_yin_setup_delete`. (The auto-correlation result vector must also be allocated beforehand, like any results vector.)

4.3 Gain and integer–float conversion

The samples are generally coded by floating-points number over 32 bits, within the range $[-1.0, 1.0]$. To convert them into 16 bits signed integers (which is the format used by some systems, like *HTK*), one can apply a simple gain of 2^{15} by multiplying every sample.

4.4 Pre-emphasis

The pre-emphasis is a simple first-order difference between a current sample and the previous one (weighted by a factor):

$$s(n) = s(n) - f \times s(n-1)$$

It is often used for voice analysis with a factor of 0.97 as it reduces the low frequencies while raising the high frequencies, thus amplifying the contrast.

4.5 Windowing

If the samples vector-size is known, it is possible to pre-calculate the weights that will be used to apply a given function. The function `rta_window_hamming_weights` computes a *Hamming* window while the function `rta_window_hann_weights` computes a *von Hann* window. These (or any weights vector) can be applied with the `rta_window_apply` function. The `_in_place` post-fixed functions change the input samples vector values directly.

If the samples vector-size is not known in advance, one can still apply the window using the `rta_window_rounded_apply` function. There is no interpolation, then. The weights vector indexes are simply scaled and rounded: this is efficient but the rounding error may be unacceptable if the size of the weights vector is too small comparing with the samples vector size. It is also possible to compute and apply a window on the fly, with the functions `rta_window_hann_apply` and `rta_window_hamming_apply`.

4.6 Linear predictive coefficients (*LPC*)

The `rta_lpc` function calculates the linear predictive coefficients (*LPC*) for a samples vector, using an auto-correlation and a *Levinson-Durbin* decomposition. Note that the *LPC* order is one value less than the *LPC* size.

The first *LPC* coefficient is always 1 and is often replaced (e.g. in *HTK*) by the prediction error, which gives the energy of the samples vector. If the *LPC* is computed on overlapping samples vectors, they are often windowed in order for the coefficients to evolve smoothly from frame to frame, and the energy is then reduced (by a constant factor depending on the window).

5 Descriptors based on the power spectrum

Some descriptors are based on the power spectrum of a samples vector. The samples vector is first windowed. Then a real *Fourier* transform is applied, giving a complex spectrum. The power spectrum is the square of the magnitude of the complex spectrum.

5.1 Real Fourier transform

Before computing a real *Fourier* transform, a new real *Fourier* transform setup must be allocated and filled with the `rta_fft_real_setup_new` function, with the type `real_to_complex_1d`. It will be released by the function `rta_fft_setup_delete`. The function `rta_fft_execute` applies the *Fourier* transform to a samples vector.

The transform size must be a power of 2. If the transform size is bigger than the actual samples vector, it is then padded with zeros.

By convention (e.g. *HTK*), no scale is applied to this direct transform. The inverse of the transform size can later be applied to the inverse transform in order to obtain the identity transform.

5.2 Complex spectrum to power spectrum

The power spectrum is the square of the magnitude the complex spectrum. Its size is half the size of the *Fourier* transform plus one (the last element corresponds to the *Nyquist* frequency). To get a correspondence between the power spectrum index and the corresponding frequency, one can apply a simple ratio between the maximum frequency (which is half of the sample-rate) and the maximum index (which is half of the *Fourier* transform size, as all the indexes start at 0):

$$frequency = index \frac{sample\_rate}{transform\_size}$$

5.3 Statistical moments

The statistical moments can be computed from any samples vector, as long as the weights are positive, as they represent the probability of the random variables to appear. The power spectrum conforms to this, as any value is positive, but not the amplitude spectrum (if not translated above 0). The same applies for the moments of the mel bands.

The moments are calculated over the indexes and weighted by the input values. They will be normalised by the sum of the input values in order to get the indexes probability. Note that all moments (but the first) are centred. Any moment above the second can be standardised.

The moments described hereafter describe the power spectrum moments.

5.3.1 Centroid

The spectral centroid is the first moment over the indexes weighted by the vector of power spectrum values. It is computed by `rta_weighted_moment_1_indexes`. The result unit is *index* (of the power spectrum, starting at 0). This function returns also the input sum as it can be used in further calculations.

$$m_1 = centroid = \frac{\sum_i i \times input(i)}{\sum_i input(i)}$$

5.3.2 Spread and deviation

The spectral spread is the second central moment over the indexes weighted by the vector of power spectrum values. It is computed by `rta_weighted_moment_2_indexes`. The result unit is $index^2$ (of the power spectrum, starting at 0).

$$m_2 = spread = \frac{\sum_i (i - centroid)^2 \times input(i)}{\sum_i input(i)}$$

The standard deviation is $std = \sqrt{spread}$.

5.3.3 Skewness

The spectral skewness is the third standard central moment over the indexes weighted by the vector of power spectrum values. It is computed by `rta_std_weighted_moment_3_indexes`. The result is without unit.

$$m_{3std} = skewness = \frac{\sum_i (i - centroid)^3 \times input(i)}{std^3 \sum_i input(i)}$$

5.3.4 Kurtosis

The spectral kurtosis is the fourth standard central moment over the indexes weighted by the vector of power spectrum values. It is computed by `rta_std_weighted_moment_4_indexes`. The result is without unit.

$$m_{4std} = kurtosis = \frac{\sum_i (i - centroid)^4 \times input(i)}{std^4 \sum_i input(i)}$$

Note that the kurtosis is often defined as the fourth cumulant divided by the square root of the variance, which gives $kurtosis = \frac{m_4}{std^4} - 3$. This function does not include the "-3" term.

6 Descriptors based on the mel bands

The mel scale can be derived from the frequencies in hertz. The conversion functions are in `rta_mel.h`. They are based on two slightly different formulas, according to *HTK* or the *Auditory Toolbox* with the respective suffix `_htk` or `_slaney`.

The power spectrum is integrated into several bands, according again to *HTK* or the *Auditory Toolbox*. The integration window peak is 1 for *HTK*, while the sum of any channel is 1 for the *Auditory Toolbox*.

In order to reproduce the results of one of these tools, one must obviously choose the desired variant among the whole computation process.

6.1 Mel bands

The mel bands integration is done in the magnitude (abs) or the power (abs^2) domain, using respectively the function `rta_spectrum_to_bands_abs` or `rta_spectrum_to_bands_square_abs`. This respectively gives

$$mel\_bands = weights\_matrix \times spectrum$$

or

$$mel\_bands = (weights\_matrix \times \sqrt{spectrum})^2$$

The latter is the default (for *HTK* and *Auditory Toolbox*) but it involves more computation.

The matrix to multiply the power spectrum vector with, in order to obtain the mel bands, must be computed beforehand, using the `rta_spectrum_to_mel_bands_weights` function.

6.2 Mel-frequency cepstral coefficients (*MFCC*)

First, the logarithm of the mel bands values is taken. In order to avoid $log(0)$, one can add a very small value to the mel bands values before taking the log.

Then a cepstrum is computed for the log of the mel spectrum, using a discrete cosine transform (*DCT*) of type II. *HTK* uses an orthogonal but not unitary transform, while the *Auditory Toolbox* uses an orthogonal and unitary transform. First, a weights matrix is constructed with the function `rta_dct_weights` and it is then applied to the mel bands vector using the function `rta_dct`. The coefficients obtained are the *MFCC*.

The *MFCC* as any *DCT* coefficients, are ordered by the order of importance to model the spectrum. However, one can need to modify them (for visualisation or further processing) using the liftering functions in `rta_lifter.h`. These functions are provided for the *HTK* or *Auditory Toolbox* compatibility.

6.3 *Delta* and *delta-delta MFCC*

The function `rta_delta` computes a simple linear slope on a sequence of fixed-rate sampled data (the frames). The *delta* values correspond to the mid-point frame (which is not used, by the way). *It means that the delta values are not those of the last frame.* Considering the filter-size, which is the size of the sequence taken into account, the delay (in frames) introduced is half of the filter-size (rounded down as it is always odd).

Note that the *HTK* `DELTAWINDOW` variable is not the same as the filter-size (the same applies for the `ACCWINDOW` variable):

$$filter\_size = 1 + 2 \times DELTAWINDOW$$

Beforehand, a matrix of weights to multiply the sequence vector with is constructed by the function `rta_delta_weights`. A normalisation factor, computed by the function `rta_delta_normalization_factor` gives *delta* values, which are independent of the filter-size. The normalisation factor can be applied directly to the weights matrix but the rounding errors may be unacceptable when using the simple floating-point precision.

Applying the *delta* computation again gives the *delta-delta* values, *adding a new delay of half of the delta-delta filter-size.*