FASTLab Inc.
220 Santa Anita Rd.
Santa Barbara, California, 93105, USA
http:www.FASTLabInc.com

FASTLab Music Analysis Kernel Library (FMAK)
Version 2.0
Technical Documentation

Stephen Travis Pope
Alex Kouznetsov
September, 2004

TABLE OF CONTENTS

Overview

The FASTLab Music Analysis Kernel (FMAK) is a software library for music content analysis and feature extraction. The first-stage signal analysis derives a set of approximately 40 features; these include both time-domain and spectral-domain features. The first-level features are then used by the subsequent analysis phases, which derive higher level musical and recording/production features. These analysis processes can also request additional signal-level features to be derived.

The FMAK API consists of the external specifications of a hierarchy of C++ classes for processing sound and interacting with relational databases.

The basic operations provided by the FMAK software are:

1: Detailed feature extraction (time-domain, frequency-domain, peak tracking, etc.)

2: Multi-stage hierarchical time-segmentation

3: Segment-based data averaging and pruning

Each of these steps is presented in more detail below.

FMAK Data Structures

The FMAK API defines a hierarchical set of data abstractions (classes), organized as follows:

FeatureDatum (FD) -- a single value, typically a float or double.

FeatureVector (FV) -- an array of FeatureData that knows its size (e.g., a spectrum array).

FeatureTable (FT) -- a data structure of FeatureData and FeatureVectors, related to the same window of time (contains RMS data, spectral info, etc.) -- this is sometimes called a feature vector in the literature).

FeatureCollection (FC) -- a time-sorted collection of FeatureTables for a musical selection, including song meta-data such as title and artist and segmentation information; this is what we store per-song and use for clustering.

FeatureSet (FS) -- the largest database type, a set of feature collections for a group of songs; these are used to represent groups of FCs in the database, e.g., a genre.

Each of these data structures is presented in more detail below, full C++ code listing can be found in Appendix A. The lists below are the central variables of the FeatureTable (FT) and FeatureCollection (FC) classes.

A FeatureTable object is used for a single window of time, or for longer averaged windows. The general-purpose data members of a feature table are:

General and Time-domain features

Duration (number of windows averaged together)

Peak sample, average RMS level

Lo- and hi-frequency filtered signal RMS values

Dynamic range of sub-windows

Stereo width, surround separation

Segment cue points and durations

Count of Zero Crossings

Tempo estimate

FFT and pitch domain features

FFT spectrum (1024 ptrs + 1-oct (10 bands) and 2.5-oct (4 bands))

Count of FFT spectral peaks

FFT statistics: spectral centroid, flatness measure, spectral variety

Spectral track statistics (births / deaths of tracked partials)

Special features of high-frequency information

Estimate of the bass pitch

LPC

LPC predictor coefficients

LPC residual (noise) level

Count of LPC spectral peaks / formants

Formant track statistics (births / deaths of tracked formants)

FWT

Discrete Wavelet Transform coefficients

Beat structure derived from the DWT

DWT-derived noise estimate

The initial analysis fills in most of the FT values, but some properties are only meaningful for longer running-average feature matrices.

A FeatureCollection object is used for a musical selection (a song); its most important components are:

Song name, artist, title, track, album, year, duration

Genre(s) -- weighted list of related genres

Windowed and 1-sec average FeatureTable data for song

Song-Average / Peak / Typical FeatureTable data

Single-note FeatureTable for some selected “solo instrument” window (optional)

Segment-length vector and weights

Tempo and time-domain characteristics

FFT / LPC peak-tracking statistics

Quiet, loud, and repeat sections

Fade-in / out times

Standard Analysis Stages

A typical application will use FMAK objects for constructing a music database in these stages:

1: A song is loaded, converted to a 2- or 5-channel sample buffer. Stereo material is converted to sum and difference channels, and 5.1-channel surround sound material is converted into sum, L / R separation, F / R separation, center, and LFE channels.

2: The standard analyzer set (RMS, FFT, LPC, and Wavelet processing) is run on each window of the data buffer, creating a large feature collection full of complex feature tables.

The various analyzer stages may have different window and hop sizes, e.g.,

a 4-segment RMS envelope and peak value every 1024 frames,

a 1024-point FFT every 8192 frames,

a 24-pole LPC every 16384 samples, and

a 1024-coefficient fast Wavelet transform (FWT) every 16384 samples.

The analysis driver also creates running 1-second “weighted peak” average feature data, (which is important to the segmenter), as well as per-song peak and average feature tables.Peak extractors and trackers are run on the FFT and LPC data to gather time-varying track birth / death statistics.

3: The segmenter uses a trained distance metric (weighting) function to segment the data into “sections” or “verses.” Several distance metrics are used on the running-average data to locate significant changes (peak deltas, segment transition points). Starting with the time-averaged data, we sub-divide segments hierarchically to accurately locate the transition points. The segment cues are related to one another to discover same-length sections (where present). The cue list is weighted and pruned to 2-10 points, 0 points (no change) is a special case. The segment-length vector is added to the feature collection as an important new feature.

4: The feature data is averaged per-segment and the collections are “pruned” This leaves us with a set of simplified time-keyed analysis frames in the greatly reduced feature collection. We also add song-average, typical verse peaks, etc. to the feature collection. We may also select a “solo instrument note” and maintain its detailed feature matrix.

5: Several database records are stored for the song's feature data (1 FC and several FTs).

FMAK Library Calls

User applications have access to the core functions of the FMAK system through the Driver class. One creates an instance of a driver giving it the analysis window sizes and hop sizes for the separate analysis stages. The default values for these parameters are given in Appendix B of this document.

The Driver constructor has the following function signature.

```
                                    // Constructor
Driver(unsigned rmsW, unsigned rmsH, unsigned rmsSubW,
                unsigned fftW, unsigned fftL, unsigned fftH,
                unsigned lpcW, unsigned lpcO, unsigned lpcH);
```

Given a Driver instance, one calls the analysis engine using the process() method, which is available in several versions as shown in the function prototypes below.

```
                                    // The main fcn calls -- do windowed analysis loop on a buffer
    void process_song(Buffer & theSong);
    void process_song(Buffer & theSong, char * genre);
    void process_song(Buffer & theSong, char * fileName, char * artist, char * title,
                      char * album, char * genre);
```

The Driver object has a FeatureCollection as a public data member, so that an application can access it after running the song processing. There are also utility functions to store the Driver's feature collection to a file directory (for debugging) or to a PostgresSQL database. The Driver methods for this are as follow.

```
    void write_fc_2_database();                          // store features to PostgresSql RDBMS
    void write_fc_2_disk(string directory, bool format); // Format = true for flat binary file,
                                                         // false for ASCII files in a directory
```

For more examples of the use of the Driver object, see the analysis driver or the EMA application source code.

FMAK Library Analysis Sequence

Analysis Driver

A driver is required to run FMAK library, since it is only a library. The FMAK Analysis Driver utility provided as a part of FMAK package performs the necessary operations to load a sound file into memory, create the necessary data structures, call the FMAK library, and store the results into the database. More information on the FMAK driver can be found in "FMAK Tools and Utilities: Technical Documentation". The Analysis Driver is often run in batch mode over large sets of sound files, representing many CDs worth of material, however, single song analysis is supported as well.

Running FMAK Library

The software framework uses the concepts of the CREATE Signal Library (CSL), so FMAK analyzers are akin to CSL's FrameStreams; all implement the work-horse method next_feature_vector(Buffer & input, unsigned offset, unsigned window, FeatureTable & fm) to read a buffer of input samples and fill in items in the given feature table. The process can be customized as to the windowing used in the analysis, which determines the structure and volume of the resulting FeatureCollection data (the list of feature table data structures for the song). The analysis first pass collects multi-resolution analysis and statistics, which generates a hierarchical time-keyed feature collection where the most-frequent frames are RMS data only (shown as "r" in the figure below), and a sub-set of analysis frames has the full (FFT ("f"), LPC ("l"), FWT ("w")) analysis data filled in.

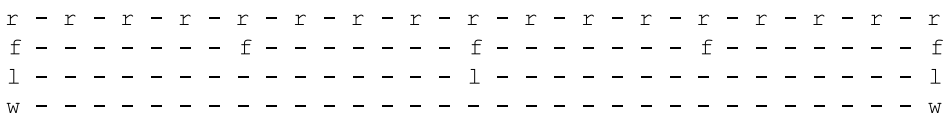

Figure 1: Windowing with different hop-sizes for initial analysis
(r = RMS analysis, f = FFT, l = LPC, w = Wavelet)

While computing this sequence, we also maintain 1-second averaged feature tables, as well as average and peak tables for the entire song.

Peak Detection and Tracking

After the first-pass windowed analysis, spectral / formant peak detectors and inter-frame peak trackers are run on the FFT and LPC spectra, locating notes and chords, and deriving track birth / death statistics that are useful (along with the RMS data) for segmentation.

Segmenting

The segmenter locates “cut points” in sequence; it uses a weighted (RMS / spectral / track) inter-segment distance function and delivers a list of segment durations for verses, phrases, beats, etc. There are several approaches possible for the segmenter, and FMAK tries them in order to find a song segmentation with a high level of confidence. The segmenter starts with the 1-second moving averages, and locates the peaks in the inter-frame difference function, hopefully finding a reasonable number (2 - 10) of them. It then uses the precise windowed RMS envelopes to guess the exact time of each segment boundary. Taking this array of times, we group the segments and search for a common segment length (in the range of 10 - 45 seconds), which might correspond to a verse or section of a musical form.If we find a verse structure, we finally gather some statistics about the number and regularity of the verses, and the prelude / postlude (or fade-in / out, or intro / coda) sections.

Pruning and Reduction of FeatureCollections

The final analysis phase is to use the list of song sections to average and then trim out unused feature table structures from the (large) song feature collection. Using the segmentation data, we can choose to keep average, peaks, and / or “typical verse” feature tables for the song's sections, and create a new section-weighted average for the whole song. We end up with a feature collection that has a fixed set of tables for the song, and store it to the relational database as several records in various database tables.

Appendix A. Analysis Data Structure Details

The code below is an excerpt from the actual class definitions for the two primary objects used in the FMAK data base: feature tables and feature collections.

```
// A FeatureDatum's just a float

typedef float FeatureDatum;              // short-hand for default feature data type (float)

// A FeatureTable is used to hold all the features for a given time-range of sound
// (might be the whole song). This is what other systems refer to as the feature vector.

// This is a record-like object, with many public data members.

class FeatureTable {                     // FeatureTable is a root object (no parents)

public:                                          // Data members (instance variables)
    float mTimeStamp;                            // When do I start?
    float mTimeDur;                              // How long a time-span do I represent?

                                         // Time-domain features
    unsigned int mRMSWindowSize;                 // Size of RMS window
    FeatureDatum mRMS;                           // Rectangular-windowed RMS amplitude
    FeatureDatum mPeak;                          // Max sample amplitude
    FeatureDatum mLPRMS;                         // RMS amplitude of LP-filtered signal
    FeatureDatum mHPRMS;                         // RMS amplitude of HP-filtered signal
    size_t mZeroCrossings;                       // Count of zero crossings
    FeatureDatum mDynamicRange;                  // RMS dynamic range of sub-windows
    FeatureDatum mPeakIndex;                     // RMS peak sub-window index
    FeatureDatum mTempo;                         // RMS/FWT instantaneous tempo estimate
    FeatureDatum mTimeSignature;                 // Time signature guess
    FeatureDatum mBassPitch;                     // Bass pitch guess in Hz
    unsigned int mBassNote;                      // Bass note (MIDI key number) guess
    FeatureDatum mBassDynamicity;                // Bass note dynamicity (running histogram)

                                         // Spatial features
    FeatureDatum mStereoWidth;                   // L/R difference
    FeatureDatum mSurroundDepth;                 // Front/Surround difference
    FeatureDatum mCenterDistinction;             // Center vs. L/R sum difference

                                         // Frequency-domain features
    unsigned int mFFTWindowSize;                 // Size of FFT window
    FtVector mSpectrum;                          // Hanning windowed FFT data (1024 points)
    FtVector mReducedSpectrum;                   // 1-octave FFT data (10-12 points)
    FtVector mBandSpectrum;                      // 2.5-octave FFT data (4 spectral bands)
    FtVector mCenterSpectrum;                    // 2.5-octave center-channel (if 5.1-ch)
    FPartialVector mSpectralPeaks;               // List of major spectral peak indeces
    FPartialVector mSpectralTracks;              // List of tracked peak frequencies
    FeatureDatum mSpectralCentroid;              // Spectral centroid measure
```

```
    FeatureDatum mSpectralSlope;            // Spectral slope measure
    FeatureDatum mSpectralVariety;          // Inter-frame spectral variety measure

                                            // Hi-frequency properties
    FeatureDatum HiFreqBalance;             // Relative HF level
    FeatureDatum HiFreqVariety;             // HF inter-frame spectral variety
    FeatureDatum HiFreqCorrelation;         // Level of correlation between HF and
                                            // audio-band tracks
    FeatureDatum mSTrackBirths;             // Spectral peak track births
    FeatureDatum mSTrackDeaths;             // Spectral peak track deaths

                                            // LPC features
    unsigned int mLPCWindowSize;            // Size of LPC window
    FtVector mLPCcoeff;                     // LPC data (spectrum or NULL)
    FPartialVector mLPCFormants;            // List of LPC formant peaks
    FPartialVector mLPCTracks;              // List of tracked LPC formants
    FeatureDatum mLPCResidual;              // LPC residual level (noisiness)
    FeatureDatum mLPCPitch;                 // Pitch estimate
    FeatureDatum mLTrackBirths;             // LPC formant peak track births
    FeatureDatum mLTrackDeaths;             // LPC formant peak track deaths

                                            // Wavelet-domain (Fast Wavelet Transform) features
    FtVector mWaveletCoeff;                 // FWT coefficient or NULL
    FtVector mWTNSpectrum;                  // Reduced FWT HiFreq noise spectrum
    FtVector mWTTracks;                     // List of tracked FWT peaks
    FeatureDatum mWTNoise;                  // FWT noise estimate

                                            // Segment Traning Data
    float mTrainingDFValues[4];             // one value per window for each function
                                            // 1. gausian bump at known transition
                                            // 2. square pulse at known transition
                                            // 3 and 4 TBD
};


// A FeatureCollection is a list of FeatureTable structures stored for various times in a song.
// This is what gets stored in the database and used by segmenters, classifiers, etc.

class FeatureCollection : public RefCounted {

    public:                 // public data members

                                                        // Per-selection features
    const char * mName, * mArtist, * mTitle, * mAlbum;  // track metadata
    unsigned mVersion, mTrack, mYear, mSampleRate;      // DB track data
    Genre * mGenre;                                     // primary genre match
    FeatureDatum mDuration;                             // float dur
    unsigned mSizeInWindows;                            // # of rms windows
```

```
                        // Main vectors of feature matrices = core data
                        // (may actually be empty, if the derived data is filled in)
    FeatureTableList mWindowData;   // collection of tables for every window (10-per-sec)
    FeatureTableList mOneSecData;   // collection of tables for the 1-sec average

                                    // Averaged or reduced data sets
    FeatureTable mAvgData;              // Averaged FeatureTable data for the song
    FeatureTable mWeightedData;         // "Weighted" (peak/thresholded) data
    FeatureTable mTypicalData;          // "Typical verse" (selected by the segmenter)
    FeatureTable mSoloData;             // "Solo verse" (or solo instrument note
    FeatureTable mNoteData;             // Matrix of a single solo instrument note

                                    // Segmentation data
    FtVector mSegmentPoints;            // Vector of segment cue points
    FeatureDatum mSegmentWeight;        // Confidence weight of the segmentation
    FeatureDatum mVerseLen;             // Most common segmentation interval
    FeatureDatum mFirstVerse;           // Duration of initial segment

                                    // Segment/envelope statistics
    FeatureDatum mQuietSections;        // % of long-ish quiet sections
    FeatureDatum mLoudSections;         // % of long-ish tutti sections
    FeatureDatum mRepeatSections;       // % of repeated sections
    FeatureDatum mFadeIn;               // dur of initial fade-in
    FeatureDatum mFadeOut;              // dur of final fade-out

};
```

Appendix B. Default Analysis Parameters

The following code is included at the end of the main header file FMAK.h; it sets the default analysis parameters that are typically used by the batch analysis processor in building databases.

```
//
// Default window sizes, transform lengths, and inter-window hop sizes
//

    // Windowed RMS averages

#define DEFAULT_RMS_WIN_SIZE 1024       // beat analyzer base window size
#define DEFAULT_NUM_RMS_SUB_WINS 1      // number of sub-windows for dyn-range and
                                        // peak data
#define DEFAULT_RMS_HOP_SIZE 1024       // inter-window hop size

    // Windowed FFT spectra

#define DEFAULT_FFT_WIN_SIZE 1024       // fft data win len
#define DEFAULT_FFT_LEN 1024            // fft length
#define DEFAULT_FFT_HOP_SIZE 1024       // inter-window hop size

    // Windowed LPC coefficients

#define DEFAULT_LPC_WIN_SIZE 1024       // lpc window
#define DEFAULT_LPC_ORDER 24            // lpc filter order
#define DEFAULT_LPC_HOP_SIZE 1024       // inter-window hop size

    // Windowed FastWaveletTransform spectra

#define DEFAULT_FWT_WIN_SIZE 1024       // FWT window size
#define DEFAULT_FWT_LEN 512             // FWT size
#define DEFAULT_FWT_HOP_SIZE 1024       // inter-window hop size
```

Appendix C: Code Examples

These code examples demonstrate the use of the FMAK class library for off-line database development and for run-time analysis within an interactive application.

Analysis Driver main Loop

```
// SPECIFICATION

//
// The analysis driver class -- runs through a stored sample array...
//      (does not know about file IO; this is all in memory for now)
// Note that (for historical reasons), this class doesn't obey the typical
// naming convention, i.e., data member names don't start with 'm;'
//

class Driver {

public:                                         // window sizes and hops are public for easy access
                                                // even though these are normally set in the constructor
        unsigned rmsWin, rmsHop, rmsSubWin,
                 fftWin, fftLen, fftHop,
                 lpcWin, lpcOrd, lpcHop,
                 fwtWin, fwtLen, fwtHop,
                 ioWinSize, ioHop;
                                                // the master feature collection is also public
        FeatureCollection * features;
                                                // Constructor takes the relevant window/hop sizes
        Driver(unsigned rmsW, unsigned rmsH, unsigned rmsSubW,
                        unsigned fftW, unsigned fftL, unsigned fftH,
                        unsigned lpcW, unsigned lpcO, unsigned lpcH);
        virtual ~Driver();
                                        // The main fcn call -- do windowed analysis loop on a buffer
        void process_song(Buffer & theSong);
        void process_song(Buffer & theSong, char * genre);
        void process_song(Buffer & theSong, char * fileName, char * artist, char * title,
                        char * album, char * genre);

                                                // Data storing functions
#ifdef STORE_IN_DATABASE
        void write_fc_2_database();
#endif
        void write_fc_2_disk(string directory, bool format);// true for flat, false for directory

protected:
                                                // FeatureCollection processors used by process_song
        void main_analysis_loop(Buffer & theSong);
        bool refine_features();                 // second-pass analysis/tracking
        void segment_collection();              // run hierarchical segmenter
```

```
        void adjust_time();                 // track tempo and adjust peak times
        void prune_segments();              // reduce feature collection and do segment statistics

};

// IMPLEMENTATION

// Set up a Driver

Driver :: Driver(unsigned rmsW, unsigned rmsH, unsigned rmsSubW,
                        unsigned fftW, unsigned fftL, unsigned fftH,
                        unsigned lpcW, unsigned lpcO, unsigned lpcH)
                : rmsWin(rmsW), rmsHop(rmsH), rmsSubWin(rmsSubW),
                        fftWin(fftW), fftLen(fftL), fftHop(fftH),
                        lpcWin(lpcW), lpcOrd(lpcO), lpcHop(lpcH) {
        // anything to do here??
}

Driver :: ~Driver() { }

// Analysis processing work functions

// Main analysis loop that creates a hierarchical feature collection with
//      RMS, FFT, LPC, and FWT data, as well as moving 1-sec moving average and peak

void Driver :: process_song(Buffer & theSong, char * fileName, char * artist, char * title,
                                char * album, char * genre) {
        unsigned numFrames = theSong._numFrames;
        float durInSecs = (float) numFrames / (float) CGestalt::sample_rate();
        unsigned numWindows = static_cast<size_t>
                                ( ( static_cast<float>( numFrames ) / rmsHop ) + 1 );

                                        // Create the master feature collection (this'll be large)
        features = new FeatureCollection(fileName, artist, title, album, CGestalt::sample_rate(),
                                numWindows, durInSecs, rmsWin, fftWin, lpcWin);
        features->mGenre = new Genre(genre);

                                        // Print settings
        printf("\tAnalysis of %.2f sec, %d windows -- rh %d fh %d lh %d",
                        durInSecs, numWindows, rmsHop, fftHop, lpcHop);
        fflush(stdout);
                                        // run the main window loop
        main_analysis_loop(theSong);
        printf("\n");
                                        // run the 2nd-stage analysis tasks
        refine_features();
                                        // Tempo tracking and exact time adjustments
        cout << "\tTracking tempo, correcting timing info" << endl;
                                        // track tempo and adjust RMS peaks
        adjust_time( * features);
```

```
                                        // Segment Training Data
        add_training_data( * features);
                                        // Segmentation
        cout << "\tDoing segmentation" << endl;
        segment_collection( * features);
                                        // Prune: average per-segment, create song average
        cout << endl << endl << "\tPruning data" << endl;
                                        // Reduce the feature collection and do statistics
        prune_segments( * features);
}

// work loop for 1st-stage analysis

void Driver :: main_analysis_loop(Buffer & theSong) {
                                        // hop size multiples of various analyzers
        size_t fftRecordDataPeriod = (size_t) (((float) fftHop / (float) rmsHop) + 0.5);
        size_t lpcRecordDataPeriod = (size_t) (((float) lpcHop / (float) rmsHop) + 0.5);
        size_t oneSecondAvgPeriod  = (size_t) (((float) CGestalt::sample_rate() / (float) rmsHop));
        size_t fwtSkip = 0;             // FWT not turned on yet
        unsigned numWindows = features->mSizeInWindows;

                                        // create the analyzers
        BeatAnalyzer beatAnalyzer(rmsWin, rmsSubWin);
        SpatialAnalyzer spatialAnalyzer(rmsWin);
        SpectralAnalyzer spectralAnalyzer(fftWin, fftLen);
        LPCAnalyzer lpcAnalyzer(lpcWin, lpcOrd);
        PitchAnalyzer pitchAnalyzer(rmsWin);
        WaveletAnalyzer  waveletAnalyzer(fwtWin, fwtLen);
        PitchAnalyzer::initialize_MIDI();
#ifdef FMAK_DEBUG
        cout << "\tProcessing loop" << endl;
#endif
                                    // initialize iteration data
        size_t oneSecIndex = 0;
        FeatureTable * pOneSecDataAverage = features->mOneSecData[oneSecIndex];
        FeatureTable songAverage, songPeaks;

                                    //// MAIN ANALYSIS LOOP
        for (size_t windowIndex = 0; windowIndex < numWindows; windowIndex++) {
#ifdef EMA_APP                      // update EMA progress bar
                if ((windowIndex % 50) == 0)
                        set_EMA_percent_complete((float) windowIndex / (float) numWindows);
#endif
//#define DEBUGGING_ANALYSIS
#ifdef DEBUGGING_ANALYSIS
                if ((windowIndex % 1000) == 0) {
                        printf(" .");
                        fflush(stdout);
                }
#endif                                  // get the current FT pointer
```

```
                FeatureTable * pCurrentFeatureTable = features->mWindowData[windowIndex];
                                        //// run the analyzers one by one ////
                                        // beat analyzer
                beatAnalyzer.next_feature_vector(theSong, 0, windowIndex, *pCurrentFeatureTable);
                                        // beat analyzer
                spatialAnalyzer.next_feature_vector(theSong, 0, windowIndex,
                                *pCurrentFeatureTable);
                                        // bass pitch analyzer
                pitchAnalyzer.next_feature_vector(theSong, 0, windowIndex, *pCurrentFeatureTable);
                                        // FFT analyzer -- gathers reduced spectrum and statistics
                if (windowIndex % fftRecordDataPeriod == 0)
                        spectralAnalyzer.next_feature_vector(theSong, 0, windowIndex,
                                *pCurrentFeatureTable);
                                        // LPC analyzer -- this sets residual and peaks
                if (windowIndex % lpcRecordDataPeriod ==  0)
                        lpcAnalyzer.next_feature_vector(theSong, 0, windowIndex,
                                *pCurrentFeatureTable);
                                        // FWT analyzer
                if (fwtSkip--) {
                        waveletAnalyzer.next_feature_vector(sampleWindow, 0, windowIndex,
                                *pCurrentFeatureTable);
                        fwtSkip = fwtHop / rmsHop;
                }
                                        // Calculate running averages and peaks
                pOneSecDataAverage->StoreSumOfMutableData(* pCurrentFeatureTable);
                features->mAvgData.StoreSumOfMutableData(* pCurrentFeatureTable);
                features->mWeightedData.StoreMaxOfMutableData(* pCurrentFeatureTable);

                if (windowIndex % oneSecondAvgPeriod == 0) {
                        pOneSecDataAverage->ScaleMutableData(1 / static_cast<float>
                                        (oneSecondAvgPeriod));
                        oneSecIndex++;
                        pOneSecDataAverage = features->mOneSecData[ oneSecIndex ];
                }
        }                               // do the averaging
        features->mAvgData.ScaleMutableData(1.0f / (float)(features->mWindowData.size()));
}
```

EMA Application Analysis Interface

```
// SPECIFICATION

class AnalysisEngine {

public:
        AnalysisEngine();
                                // Run-time analysis
        bool run(Buffer & songBuffer);
```

```
                                        // Load the run-time database from a flat file
        FeatureDB & load_runtime_data(char * filename);
                                        // get the analysis feature collection as a struct (_fcs)
        void get_features();
        FeatureCollectionStruct _fcs;

protected:
        Driver * _driver;
        FeatureDB _database;
};

// IMPLEMENTATION

// Analysis Engine constructor

AnalysisEngine :: AnalysisEngine() {
                                                        // These defaults are in FMAK.h
        _driver = new Driver(DEFAULT_RMS_WIN_SIZE, DEFAULT_RMS_HOP_SIZE,
                DEFAULT_NUM_RMS_SUB_WINS, DEFAULT_FFT_WIN_SIZE,
                DEFAULT_FFT_LEN, DEFAULT_FFT_HOP_SIZE, DEFAULT_LPC_WIN_SIZE,
                DEFAULT_LPC_ORDER, DEFAULT_LPC_HOP_SIZE);
        rmsHop = DEFAULT_RMS_HOP_SIZE;

        FeatureTableStruct avgFTS, peakFTS;
        _fcs.mName = "";            _fcs.mArtist = "";
        _fcs.mTitle = "";           _fcs.mAlbum = "";
        _fcs.mSampleRate = 0;
        _fcs.mQuietSections = 0.0;
        _fcs.mLoudSections = 0.0;
        _fcs.mRepeatSections = 0.0;
        _fcs.mFadeIn = 0.0;
        _fcs.mFadeOut = 0.0;
        _fcs.mAvgData = & avgFTS;
        _fcs.mWeightedData = & peakFTS;
}

// Run-time analysis method

bool AnalysisEngine :: run(Buffer & songBuffer) {
        _driver->process_song(songBuffer);
        return true;
}

// Grap the analysis driver's features into an FCS structure

void AnalysisEngine :: get_features() {
        FeatureCollection * fc = _driver->features;
        ema::copy_FCS( & _fcs, * fc);
}
```