AUFTAKT SDK 3.2.6

by zplane.development

February 9, 2018

Contents

1 [aufTAKT] V3 Tempo and Beat Tracking SDK Documentation

1.1 Introduction

[aufTAKT] is a beat tracking algorithm that is able to perform beat tracking even in drumless material. Based on an onset detection scheme, [aufTAKT] performs beat tracking in an iterative way. That enables [aufTAKT] to follow tempo changes at least to some extend.

The processing consists of two basic steps. The preprocessing step extracts the information about note onsets or events from the audio file, and the actual processing step uses the extracted onset information for the estimation of the beat locations and thus the tempo of the input file. The results of the preprocessing steps can be saved to avoid unnecessary preprocessing if a file is opened frequently.

Note that the intention of a beat tracker is not to mark musical events but to detect the musical beats; thus, audio material with syncopes will result in a beat grid at the beats, not necessarily at the actual musical events.

The project contains the required libraries for the operating system [aufTAKT] was licensed for with the appropriate header files.

The structure of this document is as following: First the API of the [aufTAKT] library is described. The API documentation contains naming conventions, function descriptions of the C++-API. The following usage examples (available as source code for compiling the test application) give a clear example on how to use the API in a real world application. Afterwards, a short usage description of the compiled example application is given.

> *Please note that [aufTAKT] is intended to be used with complete pieces of music, not with short snippets or loops. The minimum time needed for [aufTAKT] to make a decent guess about the initial tempo is 30 sec.!*

1.2 What's new in [aufTAKT] V3

[aufTAKT] V3 features automatic downbeat and time signature detection as well as an optional "forced straight tempo" option for electronic music. Also the beat tracking engine has been enhanced. Furthermore, the interface of [aufTAKT] V3 allows to use the V2 engine alternatively.

1.3 What's new in [aufTAKT] V3.1

[aufTAKT] V3.1 adds the new service method CaufTAKT_If::SetBeatMusicOpt(.). - When set to true the preprocessing stage favours 4/4 resp. 8/8 beats. Use this when you're pretty sure that the music to be analyzed has a 4/4 beat, e.g. electronic/dance music. By default this is disabled.

1.4 API Documentation

[aufTAKT] offers an ANSI-C++-API which can be accessed via the file aufTAKT_-If.h, where the class CaufTAKT_If provides the interface for [aufTAKT]. All variable types needed are either defined in file aufTAKT_If.h or standard C++-types.

1.4.1 Memory Allocation

The [aufTAKT] SDK does not allocate buffers handled by the calling application. The input buffers have to be allocated by the calling application. The input buffer size of the OnsetDetection module shouldn't exceed a value of 16384 frames. Audio buffers are allocated as double arrays of [channels][SamplesPerChannel].

1.4.2 Naming Conventions

When talking about **frames**, the number of audio samples per channel is meant. I.e. 512 stereo frames correspond to 1024 float values (samples). If the sample size is 32bit float, one sample has a memory usage of 4 byte.

1.4.3 Example Description of Output

The image below shows a short drumloop.

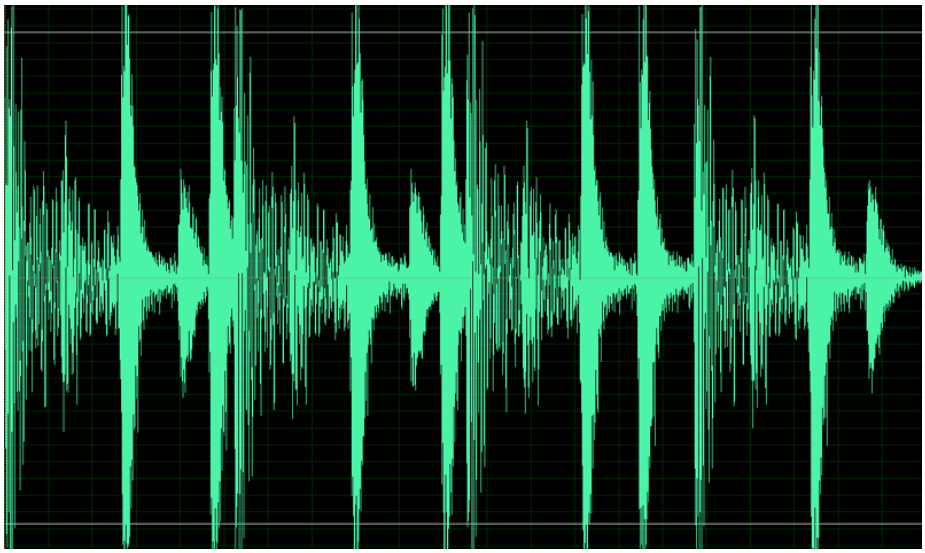

Figure 1: original drum loop

The PreAnalysis extracts the positions of the onset marks shown in the figure below. The onset marks are displayed as vertical lines within the original signal.

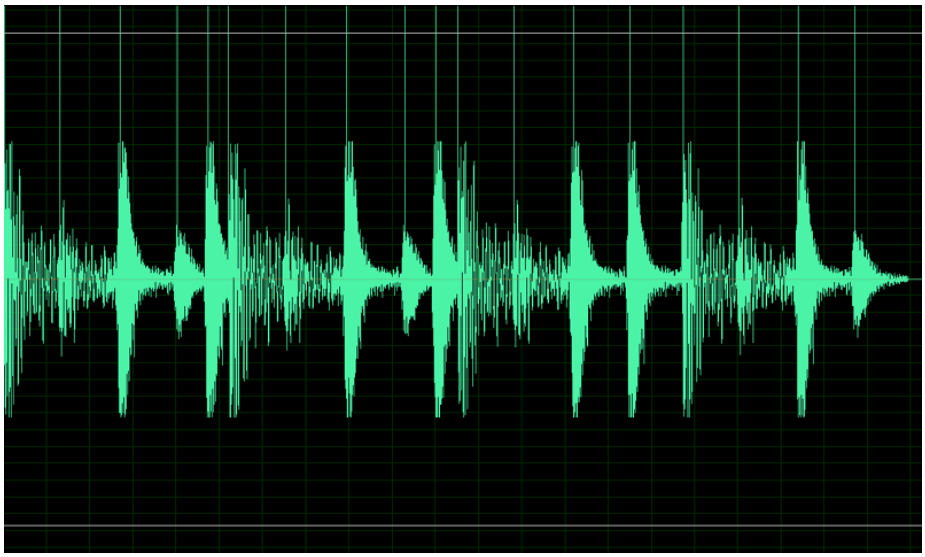

Figure 2: drum loop with onset marks

The Process extracts the positions of the estimated beat marks as shown below. The long lines represent the estimated downbeats.

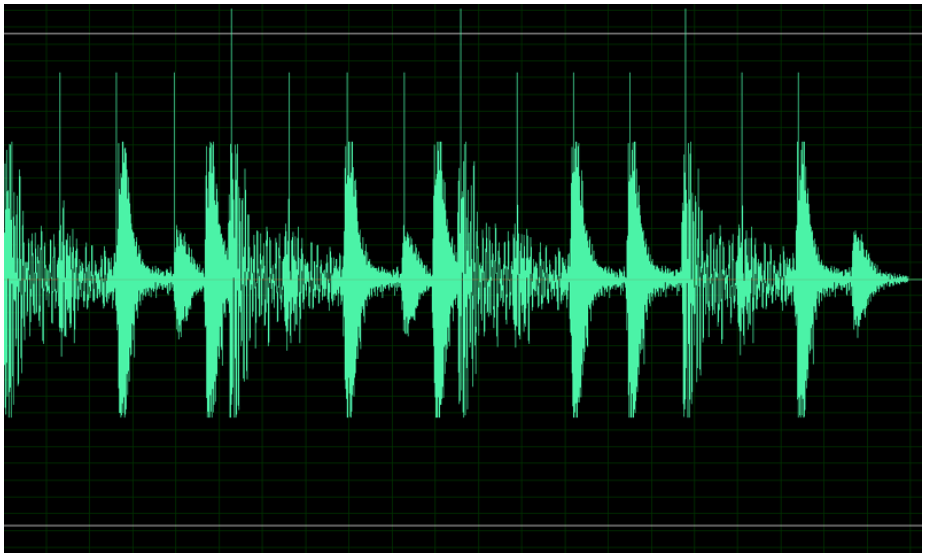

Figure 3: drum loop with generated beat marks

1.4.4 C++-API description

1.4.4.1 Required Functions

In order to do appropriate beat tracking, two steps are necessary:

- First, do the preprocessing via pushing audio data into the PreAnalysis function (or load previously calculated preanalysis data).
- Use the Process function to extract the beat location (resp. beat marks)

The following functions have to be called when using the [aufTAKT] library. Description see below.

- **CaufTAKT_If::CreateInstance (.)**

  description see below
- **either CaufTAKT_If::PreAnalysis (.) followed by CaufTAKT_If::FinishPreAnalysis() or CaufTAKT_If::SetPreAnalysisResult ()**

  descriptions see below
- **CaufTAKT_If::Process (.)**

  description see below
- **CaufTAKT_If::DestroyInstance (.)**

  description see below

1.4.4.2 Complete Function Description

1.4.4.2.1 Instance Handling Functions

- **int CaufTAKT_If::CreateInstance (CaufTAKT_If∗& pCaufTAKT_If, int iSampleRate, int iNumOfChannels, eaufTAKTVersion_t eVersion = kaufTAKT2)**

  Creates a new instance of the [aufTAKT] beat tracker. The handle to the new instance is returned in parameter pCaufTAKT_If. The sample rate and the number of channels of the audio data to be analyzed is given in parameter iSampleRate resp. iNumOfChannels. **With [aufTAKT] V3 you can choose between the V2 or V3 engine by setting eVersion to kaufTAKT2 resp. kaufTAKT3. For legacy reasons kaufTAKT2 is the default.**

  If the function fails, the return value is not 0.
- **int CaufTAKT_If::DestroyInstance (CaufTAKT_If∗& pCaufTAKT_If)**

  Destroys an instance of the [aufTAKT] beat tracker. The handle to instance is in parameter pCaufTAKT_If, which is set to 0 by this function.

  If the function fails, the return value is not 0.
- **int CaufTAKT_If::Reset ()**

  Resets all internal variables and buffers to the default state.

  If the function fails, the return value is not 0.

1.4.4.2.2 Processing Functions

- **int CaufTAKT_If::PreAnalysis (float ∗∗ppfInputBuffer, int iNumOfFrames)**

  Does the pre-processing auf the audio data in parameter ppfInputBuffer. ppfInputBuffer is a pointer array of the dimension [channels][samples]. The number of frames is given in parameter iNumOfFrames and must not exceed 16384. - This function expects to be called iteratively until there is no new audio data available. In this pre-processing step, the onset information is extracted from the audio data. The actual beat tracking uses only the resulting information, not the audio data itself. As a consequence, this function produces the most significant workload of the whole processing step. The call of this function requires the call of CaufTAKT_If::FinishPreAnalysis() if no more audio data is available.

  If the function fails, the return value is not 0.

- **int CaufTAKT_If::FinishPreAnalysis(bool bDoInitialBPMEstimate = true)**

  Has to be called after the last block of audio has been pushed to CaufTAKT_If::PreAnalysis to signal the end of pre-processing. The parameter bDoInitialBPMEstimate (default value: true) can be set to false if no initial BPM estimate is required. However, the BPM estimate is recommended for the initialization of the beat tracking step.

  If the function fails, the return value is not 0.

- **int CaufTAKT_If::Process (float fAdaptSpeed = 0, int iDownBeatPos = 0, bool bStartFromHere = false, bool bUseBPMAdaption = false)**

  Does the actual beat tracking based on the pre-analysis results (so the functions CaufTAKT_If::PreAnalysis/CaufTAKT_If::FinishPreAnalysis() or CaufTAKT_If::SetPreAnalysisResult(.) had to be called before).

  The parameter fAdaptSpeed controls the adaptation speed of the beat tracking process and has the range [1...50]. For audio files with varying tempo, choose a low value (e.g. 1), for audio files with very straight tempo, choose a high value (e.g. 40). In case of the default value (0), [aufTAKT] will use an internal estimate for the optimal value.

  The parameter iDownBeatPos is given in frames to let the user manually adjust the position of the downbeat if estimated wrongly. If 0, the internal downbeat estimate is used.

  The parameter bStartFromHere lets the beattracking run from the iDownbeatPos in both directions instead of running from the iDownbeatPos to the beginning and then back again. The former is the standard approach for V3 while the latter is the standard approach for V2. Nevertheless it is recommended not to use the parameter except a special iDownbeatPos is provided.

  The parameter bUseBPMAdaption is experimental and should not be used.

  If the function fails, the return value is not 0.

1.4.4.2.3 Result Handling Functions

- **int CaufTAKT_If::GetPreAnalysisResult (::stBeatInfoEntry ∗&pstOnsetInfo, int ∗piNumOfBeatInfoEntries, stBeatGrid ∗&pstBeatTrackState, int**

∗piNumOfGridPoints, float ∗pfBeatTrackLock, int ∗piDownBeatLocation = 0, int ∗piBarLength = 0)

After the pre-analysis, the user can get the results e.g. to store them for later usage. Since the pre-analysis is the most time-consuming task of the [aufTAKT] library, it can make sense to calculate the onsets only one time for a frequently used audio file. The memory allocation is done by the library. The entries must not be changed after this function call. The parameter pstOnsetInfo will contain the detected onset marks as the structure ::stBeatInfoEntry. The number of entries in this list is written to parameter piNumOfBeatInfoEntries. Parameter pstBeatTrackState with its size piNumOfGridPoints as well as the parameter pfBeatTrackLock are internal parameters. Their type is only defined by the API to allow platform independent storage of these internal state variables. **The - Parameters piDownBeatLocation and piBarLength are new to V3 and must be stored when using [aufTAKT] V3.** The call of this function is optional.

If the function fails, the return value is not 0.

- **int CaufTAKT_If::SetPreAnalysisResult (::stBeatInfoEntry ∗pstOnsetInfo, int iNumOfBeatInfoEntries, stBeatGrid ∗pstBeatTrackState, int iNumOfGridPoints, float fBeatTrackLock, int iDownBeatLocation = 0, int iBarLength = 0)**

  Alternatively to the preprocessing step, a previously stored onset information set can be used. The parameters correspond to the parameters of the function CaufTAKT_If::GetPreAnalysisResult.

  If the function fails, the return value is not 0.

- **bool CaufTAKT_If::IsBPMEstimateReady ()**

  This function may be called during the pre-analysis to detect, wether there is enough data to calculate a first guess of the bpm estimate via the function CaufTAKT_If::CalculateInitialBPMEstimate. This function is optional.

  The return value is true if the initial BPM estimate is ready, else false.

- **float CaufTAKT_If::GetInitialBPMEstimate ()**

  After the pre-analysis the initial BPM estimate can be requested via this function. This value should be stored with the PreAnalysisResult data.

  The return value is the BPM estimate.

- **float CaufTAKT_If::CalculateInitialBPMEstimate**

  This is an optional function allowing the calculation of an a priori initial BPM estimate during the PreAnalysis.

  The return value is the BPM estimate.

- **int CaufTAKT_If::SetInitialBPMEstimate (float fInitialBPM)**

  If the pre-analysis is avoided resp. no initial BPM estimate had been calculated or the user wants to initialize the beat tracking with its own BPM estimate, this function can be called with the new initial BPM value as parameter. The use of the function is recommended if the CaufTAKT_If::FinishPreAnalysis() function was called with its parameter set to false.

  If the function fails, the return value is not 0.

- **int CaufTAKT_If::GetNumBeatMarks (eTempoModes_t eTempoMode = kDynamicTempo)**

  After the calling CaufTAKT_If::Process, the number of calculated beats can be requested from the instance. **With V3 you can choose alternatively which beat tracking mode is to be used, either kDynamicTempo (default) or kStraightTempo. Please note, that the usage of that parameter must be consistent!**

  The function returns the number of beat marks.

- **int CaufTAKT_If::GetBeatMark (stBeatInfo *pBeat, int iIdx, eTempoModes_t eTempoMode = kDynamicTempo)**

  This function returns information about one special beat with index iIdx. The index of the beat start with 0, while the number of beat marks is returned by the function CaufTAKT_If::GetNumBeatMarks. For the given index, the structure pBeat is filled with information about the beat. The structure has to be allocated by the user before. **With V3 you can choose alternatively which beat tracking mode is to be used, either kDynamicTempo (default) or kStraightTempo. Please note, that the usage of that parameter must be consistent!**

  Note that the space between the beatmarks does not correspond to the length of a quarter note but a eigth note.

  If the function fails, the return value is not 0.

- **int CaufTAKT_If::GetTimeSignatureIdx(eTempoModes_t eTempoMode = kDynamicTempo)**

  This funtion return how many beatmarks make up one bar. So if the function returns for example 4, the bar has a 4/4 time signature. V2 will always return 8 corresponding to a 8/8 time signature. **With V3 you can choose alternatively which beat tracking mode is to be used, either kDynamicTempo (default) or kStraightTempo. Please note, that the usage of that parameter must be consistent!**

- **int CaufTAKT_If::GetFirstDownBeatEstimate (int iBeatDivisor = 0, eTempoModes_t eTempoMode = kDynamicTempo)**

  This function tries to estimate the first downbeat of the detected beat marks. The return value is the corresponding index of the beat mark list.

  The function returns the index of the first downbeat. **With V3 you can choose your own iBeatDivisor which corresponds to the TimeSignatureIdx. If set to zero (default) the internal estimate aka the result of GetTimeSignatureIdx() is used. Also you can choose the beat tracking mode to be used, either kDynamicTempo (default) or kStraightTempo. Please note, that the usage of that parameter must be consistent!**

1.4.5 C++ Usage example

The complete code can be found in the example source file aufTAKTClMain.cpp.

In the first step, a handle to the instance and the results structure have to be declared:

```
// handle to beat tracker instance
```

```
CaufTAKT_If         *pCaufTAKT_IfInstance  = 0;

// structure for the results
stBeatInfo          stBeatTrackResult;
```

Then, a new instance of aufTAKT has to be created.

```
// create an instance of aufTAKT
if (CaufTAKT_If::CreateInstance (   pCaufTAKT_IfInstance,
                                    (int)pFInputFile->GetSampleRate(),
                                    pFInputFile->GetNumOfChannels(),
                                    CaufTAKT_If::kaufTAKT3) != 0)
{
    delete pFInputFile;
    delete pFOutputFile;
    return -1;
}
```

The pre-analysis step requires a while loop where the PreAnalysis function is called subsequently with a new block of audio data.

```
    // do the pre processing for the current block
    pCaufTAKT_IfInstance->PreAnalysis(ppfInputData, iNumSamplesRead);
```

Optionally, the user can ask during the pre-analysis if an a priori estimate of the tempo can be calculated by:

```
/*  if (pCaufTAKT_IfInstance->IsBPMEstimateReady())
      bReadNextFrame = false;*/

  // if an output file with clicks should be written, scale the audio
 data with -6dB
  if (pFOutputFile != 0)
  {
      for(j=0; j<pFInputFile->GetNumOfChannels(); j++)
          for(i=0; i<_BLOCKSIZE; i++)
              ppfInputData[j][i] = ppfInputData[j][i]*SCALE;


      pFOutputFile->Write(ppfInputData, _BLOCKSIZE);
  }
```

If no more audio data is available, the function CaufTAKT_If::FinishPreAnalysis() has to be called to signal the library that no more audio data is available and to prepare for the beat detection step.

```
// tell aufTAKT that the pre analysis has been finished and make it ready
   to go!
pCaufTAKT_IfInstance->FinishPreAnalysis ();
```

At this point, the pre-analysis results can be optionally stored by the user. As an example, the initial BPM estimate is shown at the shell:

```
// show initial BPM estimate
cout << "Initial tempo estimate: " << pCaufTAKT_IfInstance->
  GetInitialBPMEstimate () << endl;
```

The internal results from the pre analysis step are now used for the actual beat tracking. This is done via calling the function CaufTAKT_If::Process.

```
// do the beat tracking (based on the onsets that are stored internally)
if (pCaufTAKT_IfInstance->Process ())
```

To request the tempo estimate at the end of the file, information about the last beat mark can be requested:

```
pCaufTAKT_IfInstance->GetBeatMark ( &stBeatTrackResult,
                                    pCaufTAKT_IfInstance->GetNumBeatMarks (
  )-1);
```

The function CaufTAKT_If::GetBeatMark allows to request information about the current tempo and location for every estimated BeatMark.

To get the overall tempo call:

```
cout << "Final tempo estimate: " << pCaufTAKT_IfInstance->GetOverallTempo()
   << endl;
```

Finally, get the estimated first downbeat position and time signature according to the chosen tempo model:

```
iTimeSig        = pCaufTAKT_IfInstance->GetTimeSignatureIdx();

cout << "Time signature " << iTimeSig  << "/8" << endl;

iDownBeatIdx    = pCaufTAKT_IfInstance->GetFirstDownBeatEstimate (iTimeSig)
  ;
```

After the successful processing, the created instance can be destroyed

```
// destroy aufTAKT instance
CaufTAKT_If::DestroyInstance(pCaufTAKT_IfInstance);
```

1.5 Delivered Files (example project)

1.5.1 File Structure

1.5.1.1 Documentation

This documentation and all other documentation can be found in the directory **./doc**.

1.5.1.2 Project Files

The Workspaces, Projectfiles and or Makefiles can be found in the directory **./build** and its subfolders, where the subfolders names correspond to the project names.

1.5.1.3 Source Files

All source files are in the directory **./src** and its subfolders, where the subfolder names equally correspond to the project names.

1.5.1.4 Include Files

Include files can be found in **./incl**.

1.5.1.5 Resource Files

The resource files, if available can be found in the subdirectory **/res** of the corresponding build-directory.

1.5.1.6 Library Files

The directory **./lib** is for used and built libraries.

1.5.1.7 Binary Files

The final executable can be found in the directory **./bin**. In debug-builds, the binary files are in the subfolder **/Debug**.

1.5.1.8 Temporary Files

The directory **./tmp** is for all temporary files while building the projects. In debug-builds, the temporary files can be found in the subfolder **/Debug**.

1.6 Coding Style minimal overview

Variable names have a preceding letter indicating their types:

unsigned:	u
pointer:	p
array:	a
class:	C
bool:	b
char:	c
short (int16):	s
int (int32):	i
__int64:	l
float (float32):	f
double (float64):	d
class/struct:	c

For example, a pointer to a buffer of unsigned ints will be named puiBufferName.

1.7 Command Line Usage Example

The compiled example is a command line application that reads and optionally writes audio files in WAV format. The output file is the input file with additional beat ticks.

Since the example application has no sophisticated command line parser, the order of the arguments is crucial. The command line synposis is:

```
aufTAKTTestCL input_file [output_file]
```

1.8 Support

Support for the SDK is - within the limits of the agreement - available from:

zplane.development

katzbachstr. 21

D-10965 berlin

germany

fon: +49.30.854 09 15.0

fax: +49.30.854 09 15.5

@: info@zplane.de