Using the GVC emotion recognition software

Good Vibrations Company B.V.

October 18, 2015

This document describes how you can incorporate the GVC emotion recognition (GVCemo) software into your own apps.

Contents

1. Overview

As an app builder, you use the GVCemo library in a very straightforward manner, illustrated here with an overview of the C API:

```
// First create one or more channels...

GVCemoChannel GVCemoChannel_create (   // e.g. call this twice for a stereo recording
    double samplingFrequency;   // e.g. 8000 Hz (telephone) or 44100 Hz (CD quality)
);


// ...then feed the channel(s) some audio samples from a file, microphone or telephone...

void GVCemoChannel_feed (
    GVCemoChannel channel,   // a channel you created before
    int numberOfSamples,   // the length of (a part of) your recording
    double samples [];   // e.g. the microphone signal as coming from your sound card
);


// ...then analyze the emotions in the recorded human voice.

void GVCemoChannel_analyze (
    GVCemoChannel channel,   // a channel you created before
    int emotionFormat,   // a choice between log odds, odds, probability and percentage
    GVCemoOutput *output   // the analysis results, as defined below
);


// You receive the emotion levels in the following structure:

typedef struct {
    int numberOfValidFrames;   // reports how much of the recording contains human voice
    int numberOfFramesLost;   // will be zero if you call `analyze` often enough
    double happyLevel;   // e.g. a percentage between 0 (very sad) and 100 (very happy)
    double relaxedLevel;   // e.g. a percentage between 0 (stressed) and 100 (relaxed)
    double angryLevel;
    double scaredLevel;
    double boredLevel;
} GVCemoOutput;
```

This is basically all there is to analyzing emotions with GVCemo. Besides the C API, which you can access from all C-related programming languages, we also provide bindings for Python, R and Java. The administrative effort for you as a programmer consists of no more than including a single header file and linking with a single library file. We provide library files for many versions of Windows, iOS, MacOS X, and Linux.

This document describes the functionality of the API in detail, and provides examples of how to call the API on any platform, from any of the programming languages, and in batch applications as well as in multithreaded real-time applications.

2. How to incorporate GVCemo in your software project

Basically, incorporating GVCemo in your project involves including one header file (in C-like languages) or one module definition (in Python) into your source code, and linking one machine-code library into your executable statically (for C-like languages) or at run time (for Python, Java or R).

2.1. Target platforms

You will probably *build* your app on a modern computer. However, you may want to *deploy* (distribute) your app to users who have much older computers. It is therefore important to know that the GVCemo software is thought to be able to run on the oldest and newest iPhones, the oldest and newest Macs with MacOSX, on Windows computers with any operating system between XP and 10, and on many Linux versions. Although there is no guarantee that the algorithm is fast enough for *real-time* applications on the oldest 32-bit hardware, GVC believes that you can get batch applications to work on all those computers.

More specifically, GVC provides:

- iOS static (.a) libraries for iPhone and iPad apps:
  - 32-bit: for iOS 1.0 and later on hardware from iPhone 1 (2007–)
  - 64-bit: for iOS 1.0 and later on hardware from iPhone 5S (2013–)
- Windows object (.o) files for Windows applications:
  - 32-bit: for XP and later (2001–)
  - 64-bit: for Vista and later on 64-bit hardware (2007–)
- Windows dynamic (.dll) libraries for Python/Java/R:
  - 32-bit: for XP and later (2001–)
  - 64-bit: for Vista and later on 64-bit hardware (2007–)
- MacOSX object (.o) files and static (.a) libraries for MacOSX apps:
  - 32-bit: for MacOSX 10.0 and later (2001–)
  - 64-bit: for MacOSX 10.6.8 and later on 64-bit hardware (2010–)
- MacOSX dynamic (.so) libraries for Python/Java/R:
  - 32-bit: for MacOSX 10.6 and later (2009–)
  - 64-bit: for MacOSX 10.6.8 and later on 64-bit hardware (2010–)
- Linux object (.o) files and static (.a) libraries (32-bit and 64-bit)
- Linux dynamic (.so) libraries for Python/Java/R (32-bit and 64-bit)

2.2. Integrating GVCemo files into an iOS, Windows, Mac or Linux app

If you program in a C-like language (C, C++, Objective C, Objective C++ or C#), you include the GVCemo software in your source code by #including the header file `GVCemo.h`.

To include the GVCemo algorithm in your executable, you link with a GVCemo object file or with a GVCemo archive of object files. We will now describe how this works for four major target platforms.

For **iOS** (iPhone and iPad) apps, the easiest way to proceed is to drag both the header file `GVCemo.h` and the machine-code archive `libGVCemo_iphone.a` into the file list of your Xcode project. The archive `libGVCemo_iphone.a` contains GVCemo machine code for all iPhone and iPad versions, as well as for the iPhone and iPad simulators in Xcode. The archive is composed of the following six object files:

- `GVCemo_ios_armv6.o`, for ARMv-6 processors, i.e. for iPhone 1 to 3 (2007–2009)
- `GVCemo_ios_armv7.o`, for ARMv-7 processors, e.g. for iPhone 3GS to 5 (2009–2012)
- `GVCemo_ios_armv7s.o`, for ARMv-7s processors, i.e. for iPhone 5 (2012–2013)
- `GVCemo_ios_arm64.o`, for 64-bit ARM processors, i.e. for iPhone 5S and up (from 2013)
- `GVCemo_isimu_i686.o`, for your 32-bit iPhone Simulator
- `GVCemo_isimu_x86_64.o`, for your 64-bit iPhone Simulator

These six object files are also available separately, in case you want to make your app available to a smaller number of processors and don't like to include the superflous object files in your project.

For **Windows** apps, you include in your project the header file `GVCemo.h` as well as one of the following object files:

- `GVCemo_win32.o`, for i686 (32-bit) processors (1997–)
- `GVCemo_win64.o`, for x86-64/x64/AMD64 (64-bit) processors (2003–)

For **MacOS X** apps, you include in your project `GVCemo.h` as well as one of the following object files:

- `GVCemo_mac32.o`, for i686 (32-bit) processors, from OSX 10.6 (2009) on
- `GVCemo_mac64.o`, for x86-64/x64/AMD64 (64-bit) processors, from OSX 10.6.8 (2010) on

For **Linux** apps, you include in your project `GVCemo.h` as well as one of the following object files:

- `GVCemo_linux32.o`, for i686 (32-bit) processors
- `GVCemo_linux64.o`, for x86-64/x64/AMD64 (64-bit) processors

As GVCemo is not open source code, the usual Linux strategy of distributing a complete source-code edition of your application will not be possible.

2.3. Integrating GVCemo files into a Python project

If you program in Python, you import the GVCemo software in your source code by `import`ing the module `gvcemo.py`. To use the GVCemo algorithm at run time, you open a GVCemo dynamic library from your source code. We provide one or two dynamic libraries for each of the three major desktop platforms.

On **Windows**, you open at run time one of the following dynamic libraries, depending on whether you are running 32-bit or 64-bit Python:

- `libGVCemo_win32.dll`, for i686 (32-bit) processors
- `libGVCemo_win64.dll`, for x86-64/x64/AMD64 (64-bit) processors

On **MacOS X**, you open at run time the following dynamic library:

- `libGVCemo_mac.so`, for 64-bit processors

On **Linux**, you open at run time one of the following dynamic libraries, depending on whether you are running 32-bit or 64-bit Linux:

- `libGVCemo_linux32.so`, for i686 (32-bit) processors
- `libGVCemo_linux64.so`, for x86-64/x64/AMD64 (64-bit) processors

We support Python 2.7 as well as Python 3.4. More details, as well as example code, are given in chapter 5.

3. GVCemo functionality

This chapter outlines the semantics and behavior of the GVCemo functions. Although this chapter employs the C syntax of these functions, the information in this chapter is also relevant for programmers who employ Python, Java or R.

The prototypes for the GVCemo functions are in `GVCemo.h`. It is assumed that an `int` is a 32-bits two's complement signed integer, a `double` is a 64-bit IEEE floating-point number, the byte order is the "native" one, and the alignment of elements within structs and parameter lists is the "natural" one for C on the machine where the software will run.

3.1. Creating your channel(s)

When your GVCemo-app starts up, the GVCemo module knows of no audio channels yet. Fortunately, you can create a channel for your recording, namely with the function `GVCemoChannel_create()`, whose prototype is given in `GVCemo.h`:

```
GVCemoChannel GVCemoChannel_create (
    double samplingFrequency;
);
```

3.1.1. Sampling frequency

In the call to `GVCemoChannel_create()`, you tell the channel what its incoming samples will mean. In other words, you will have to tell the channel what its *sampling frequency* is.

The sampling frequency, or *sample rate*, of an audio recording is how often the sound is "sampled" every second, i.e. how many samples are drawn in a second from the sound pressure as measured by the microphone.

The sampling frequency of the tracks on a CD is 44100.0 Hz, i.e., the sound is represented with 44100 stereo samples per second. Most recording devices support 44100 Hz as well, and some recording devices (such as the built-in sound card on a Mac) support *only* 44100 Hz. Telephone recordings typically have a sampling frequency of only 8000.0 Hz.

3.1.2. A mono recording: one channel

To create a channel for a mono recording in GVCemo with a sampling frequency of 44100 Hz, you would do

```
    GVCemoChannel channel = GVCemoChannel_create (44100.0);
```

3.1.3. A stereo recording: two channels

To create channels for a stereo recording, you could do

```
    GVCemoChannel left = GVCemoChannel_create (44100.0);
    GVCemoChannel right = GVCemoChannel_create (44100.0);
```

You can create any number of channels, limited only by the available memory on the computer.

3.1.4. Multiple simultaneous sources of recordings

If you have recordings of a telephone conversation, with one speaker in the outgoing stream and the other speaker in the incoming stream, you do

```
GVCemoChannel outgoing = GVCemoChannel_create (44100.0);
GVCemoChannel incoming = GVCemoChannel_create (44100.0);
```

just as with stereo recordings.

3.1.5. The result

After you call `GVCemoChannel_create()`, your channel will have been fully initialized. This means that for each channel a 5-second *ring buffer* has been created for the samples that you are going to feed to your channel with `GVCemoChannel_feed()`, as described in §3.2. Typically (i.e. for a sampling frequency of 44100 Hz), this buffer takes up 2 megabytes of heap memory.

3.1.6. Restrictions on the sampling frequency

Currently, GVCemo accepts sampling frequencies between 8000.0 and 48000.0 Hz. If you need a different sampling frequency, contact your representative at GVC.

3.1.7. Error messages

You can make `GVCemoChannel_create()` fail by creating an extremely large number of channels:

```
GVCemoChannel channels [100000];
for (int i = 0; i < 100000; i ++)
    channels [i] = GVCemoChannel_create (44100.0);
```

This may print

```
Out of memory when trying to create a channel.
```

because you are trying to allocate 200 gigabytes to GVCemo.

You can also make `GVCemoChannel_create()` fail by asking for an unsupported sampling frequency:

```
GVCemoChannel channel = GVCemoChannel_create (192000.0);
```

This will print

```
A sampling frequency of 192000.000000 Hz is not supported.
GVCemo accepts only sampling frequencies between 8000 and 48000 Hz.
```

3.2. Feeding samples to a channel

After you have created your channel with `GVCemoChannel_create()`, you are ready to feed samples to it. In other words, you are ready to give your channel an array of `double` values that represent the sound pressure at the microphone as a function of time. You do this with the function `GVCemoChannel_feed()`, whose prototype is given in `GVCemo.h`:

```
void GVCemoChannel_feed (
    GVCemoChannel channel,
    int numberOfSamples,
    double samples [];
);
```

3.2.1. Samples

For each second of your recording, you typically have been given 44100 *samples* (if the sampling frequency was 44100 Hz). These values could have come directly from the microphone in your app (for a real-time recording), or from two telephone streams (incoming and outgoing), or from a sound (e.g. WAV) file on disk.

In all cases, GVCemo expects that your samples have been scaled to values between -1 and +1. That is, if your sound file gives you potential values between -32768 and +32767 (which corresponds to 16-bit quantization), then you divide the raw sample values by 32768 before feeding them to a GVCemoChannel.

You specify the samples to a GVCemoChannel in the array `samples`.

3.2.2. The number of samples to feed

The value of the argument `numberOfSamples` depends on the number of samples that you want to feed to GVCemo at a time. If you want to feed the channel called `channel` with approximately 40 milliseconds of samples, and your sampling frequency is 44100 Hz, then you can specify 176 samples:

```
    GVCemoChannel_feed (channel, 176, samples);
```

In this case, the argument `samples` should be an array that contains 176 samples, with elements numbered `samples[0]` through `samples[175]`.

If you want to feed the right channel (called `right`) with 2.0 seconds of samples, with a sampling frequency of 44100 Hz, then you do

```
    GVCemoChannel_feed (right, 88200, samples);
```

where `samples[0]` through `samples[88199]` are valid sample values.

As you see from these examples, the number of samples can vary quite a bit. Typically, you follow each call to `GVCemoChannel_feed()` with a call to `GVCemoChannel_analyze()`, which processes and analyzes the samples you have just fed (see §3.3). This means that the number of samples you feed to GVCemo typically corresponds to the granularity with which you want to retrieve emotion levels:

- If you have a real-time app that visualizes the current emotion level of the speaker, you probably want to update the visualization every 40 milliseconds or so. GVC advises you not to go below below 30 milliseconds or so, as the number of "frames" evaluated (see §3.3) will vary too much for such low analysis time steps.
- If you have an app that wants to measure average emotion levels during parts of long pre-recorded sounds, you want to feed as many samples at a time as possible. The maximum duration of sound that you can feed, however, is 3.0 seconds, because the ring buffer is 5.0 seconds long and GVCemo requires a history of 2.0 seconds to be available to it at any time. This means that if your sampling frequency is 44100 Hz, the maximum number of samples you will want to feed to a GVCemo channel with `GVCemoChannel_feed()` is 132300. If you need to average over e.g. 60 seconds, you can retrieve 20 emotion levels over 3 seconds each, and average those yourself.

These points are further illustrated in the examples of chapter 4.

3.2.3. The result

After you call `GVCemoChannel_feed()`, the ring buffer in your channel will have been updated with a number of new samples. The buffer will also have forgotten the same number of the oldest (i.e. more than 5 seconds old) samples. With these new samples, you are now ready to analyze emotions with `GVCemoChannel_analyze()`, as described in §3.3.

3.2.4. Error messages

You cannot make `GVCemoChannel_feed()` fail. With a valid GVCemoChannel and sample array, `GVCemoChannel_feed()`, or a sequence of calls to `GVCemoChannel_feed()`, is happy to feed any number of samples to the ring buffer, even if that buffer overflows.

For instance, if the buffer is 5 seconds long and therefore has room for 220500 samples, you can still feed a million samples to it with (a sequence of calls to) `GVCemoChannel_feed()`. After `GVCemoChannel_feed()` has written the first 220500 samples to the buffer, the following samples that `GVCemoChannel_feed()` writes to the buffer will start to overwrite some samples that are 5 seconds older. In the end, this means that 779500 samples will have been written but overwritten, so that 779500 samples will be lost forever and will never be analyzed. Still, it is not `GVCemoChannel_feed()` but `GVCemoChannel_analyze()` who will detect this overflow.

The reason is that in real-time applications, `GVCemoChannel_feed()` will usually be called at lock time. In a real-time application you typically have to work with two separate threads: one that does the recording and one that does the analysis. For instance, the recording thread could give you repeatedly 64 samples in a callback directly from the sound card, whereas the emotion analysis is performed every 40 milliseconds (i.e. 1764 samples) in a timer that runs in the main thread. This means that the callback should write 64 samples to your special intermediary buffer, and `GVCemoChannel_feed()` should move 1764 samples from that intermediary buffer to GVCemo, followed by calling `GVCemoChannel_analyze()`. The crucial point, now, is that the intermediary buffer is used by two threads, but that only one thread at a time should have access to it. Therefore, the callback should set a lock immediately before writing the 64 samples to the intermediary buffer, and release the lock immediately after it, and the timer function should set a lock immediately before calling `GVCemoChannel_feed()` and release the lock immediately after `GVCemoChannel_feed()` returns. Thus, to minimize the time during which you have to make the intermediary buffer unavailable to the recording thread, `GVCemoChannel_feed()` needs to be a very fast function that just copies samples from the intermediary buffer to GVCemoChannel's internal ring buffer, and certainly does not write error messages. For an example see §4.4.

3.3. Getting emotion levels from a channel

After you have fed samples to a channel, you are ready to analyze them. You do this with the function `GVCemoChannel_analyze()`, whose prototype is given in `GVCemo.h`:

```
typedef struct {
    int numberOfValidFrames;
    int numberOfFramesLost;
    double happyLevel;
    double relaxedLevel;
    double angryLevel;
    double scaredLevel;
    double boredLevel;
} GVCemoOutput;

void GVCemoChannel_analyze (
```

```
    GVCemoChannel channel,
    int emotionFormat,
    GVCemoOutput *output
);
```

3.3.1. The emotion format

You will retrieve emotion values in the fields `happyLevel`, `relaxedLevel`, `angryLevel`, `scaredLevel`, and `boredLevel` of the `output` structure. The interpretation of these values depends on the `emotionFormat` that you specify.

If you specify an `emotionFormat` of 1 (or less), the five emotion values in `output` will represent *log odds*, i.e. logarithms of odds. These are values between minus and plus infinity. For `happyLevel`, for instance, a value of $-\infty$ denotes maximal sadness and a value of $+\infty$ denotes maximal happiness, with a value of 0 being neutral. Log odds are known from logistic regression, and they are the format of choice if you need to average multiple measurements, or compare a person's emotion today with that same person's emotion tomorrow.

If you specify an `emotionFormat` of 2, the five emotion values in `output` will represent *odds*. For instance, if `emotionFormat` is 2 and the resulting `happyLevel` is 3.0, this means that the odds are 3.0 to 1 that the person is happy rather than sad (i.e. there is a 75% chance that the person is happy, and a 25% chance that the person is sad). You could easily compute the odds from the log odds as follows:

$$odds = e^{logodds}$$

but if you specify an `emotionFormat` of 2, GVCemo performs this computation for you.

If you specify an `emotionFormat` of 3, the five emotion values in `output` will represent *probabilities*. For instance, if `emotionFormat` is 3 and the resulting `happyLevel` is 0.75, the probability of the person being happy is 0.75 and the probability of the person being sad is 0.25. You could compute the probabilities from the odds as follows:

$$probability = \frac{odds}{1 + odds}$$

but if you specify an `emotionFormat` of 3, GVCemo performs this computation for you.

If you specify an `emotionFormat` of 4, the five emotion values in `output` will represent *percentages*. For instance, if `emotionFormat` is 4 and the resulting `happyLevel` is 75, the person is 75 percent happy and 25 percent sad. You could compute the percentages from the probabilities as follows:

$$percentage = 100 \times probability$$

but if you specify an `emotionFormat` of 4, GVCemo performs this computation for you.

3.3.2. Are we measuring chances or degrees?

You may have noticed in §3.3.1 that we regarded as equivalent the following two statements:

- the person has a probability of 75% of being happy and of 25% of being sad
- the person is 75% happy and 25% sad

These two statements do not mean the same thing in real life, but our algorithm does not distinguish between them, i.e. the reported emotion levels are ambiguous as to whether they refer to chances or to degrees. Here's another statement that our algorithm regards as equivalent, though it may not be in real life:

- on a scale from 0 to 100, the happiness of the person is 75

Perhaps the future will bring algorithms that can distinguish between these three interpretations of emotion levels, but that time has not come yet. Until then, the algorithms estimate either the probability that a certain emotion is present, or the degree to which that emotion is present, or any combination of the two.

3.3.3. Valid frames

In the `output` structure you will find a field `numberOfValidFrames`. This refers to the number of "frames" that the emotion analysis was based on. If this is 0, the reported emotion levels are invalid (in fact, they receive the value $-2 \cdot 10^{300}$). Therefore, you can use the reported emotion values only if the reported number of valid frames is 1 or more.

3.3.4. Lost frames

If you call `GVCemoChannel_analyze` often enough, i.e. more than once every 3 seconds, the output field `numberOfFramesLost` will be 0. A value larger than 0 means that analysis frames were lost, i.e., that not all of the samples fed have been analyzed. This is a situation of *buffer overflow*. If this occurs too often in your application, and you want to analyze all samples, you should make sure that `GVCemoChannel_analyze` is called more often, e.g., that is called for every second of speech or so.

3.4. Resetting a channel

If you want to forget all samples that you have fed to a channel, you can call `GVCemoChannel_reset()`, whose prototype is in `GVCemo.h`:

```
void GVCemoChannel_reset (
    GVCemoChannel channel
);
```

You can use this function if you want to analyze multiple recordings of which you know that they have the same sampling frequency, or if you want to analyze multiple disconnected parts of a single recording. Calling `GVCemoChannel_reset` in between the analyses ensures that your new emotion measurement is not influenced by whatever samples remain in the channel's ring buffer from an unrelated recording.

3.5. Deleting all channels

When your app finishes working on emotion analysis, you can reclaim all the memory that GVCemo uses by calling `GVCemoChannel_delete()` on your channel(s). The prototype is given in `GVCemo.h`:

```
void GVCemoChannel_delete (GVCemoChannel channel);
```

So if you analyzed a two-way conversation, you can do:

```
    GVCemoChannel_delete (incoming);
    GVCemoChannel_delete (outgoing);
```

This releases the memory used by both channels.

A clean-up with `GVCemoChannel_delete()` is not necessary in most C-like apps, because they release all memory at shut-down anyway. The clean-up may become more important, though, if you run GVCemo from another programming language such as Python (see chapter 5), Java (chapter 6) or R (chapter 7).

4. Example code in C

You can use the API both in batch mode and in real-time mode. In this chapter we give examples of both.

4.1. A full example in C: the happy sweep

The following example program computes the "emotion" in a computer-geenrated sine wave that starts with a pitch of 100 Hz and linearly moves to a pitch of 300 Hz within 20 seconds.

```
/* happy_sweep.c */
/* Computes the development of the "happiness" in a sine sweep. */

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "../API/GVCemo.h"   // or wherever you put the header file

int main (int argc, char **argv) {
    double samplingFrequency = 44100.0;
    GVCemoChannel channel = GVCemoChannel_create (samplingFrequency);

    /*
     * Create a rising sweep.
     */
    double duration = 20.0;   // Seconds
    int numberOfSamples = (int) (duration * samplingFrequency);
    double *samples = malloc (sizeof (double) * numberOfSamples);

    for (int i = 0; i < numberOfSamples; i ++) {
        double time = i / samplingFrequency;
        // the local frequency is 100 + time * 10, i.e. it rises from 100 to 300 Hz
        samples [i] = sin (2.0 * M_PI * (100+time*5.0) * time);
    }

    /*
     * Compute and report the happiness level at multiple points in time. It should rise.
     */
    printf ("Happiness:   Valid frames:   Frames lost:\n");
    double *slice = & samples [0];
    double durationOfSlice = 2.0;   // Seconds
    int numberOfSamplesPerSlice = (int) (durationOfSlice * samplingFrequency);
    for (int islice = 0; islice < 10; islice ++) {
        GVCemoChannel_feed (channel, numberOfSamplesPerSlice, slice);
        slice += numberOfSamplesPerSlice;
        GVCemoOutput output;
        GVCemoChannel_analyze (channel, 4, & output);   // Percentages
        printf ("%.3f    %d    %d\n", output.happyLevel,
            output.numberOfValidFrames, output.numberOfFramesLost);
    }
}
```

4.2. Batch mode: one channel

If you have 1000000 samples, and you want to extract happiness every 2000 samples, you can obtain a sequence of 500 emotion values between 0.0 and 1.0 in the following way:

```
GVCemoChannel channel = GVCemoChannel_create (44100.0);
double buffer [1000000];
// fill the buffer with speech, perhaps from a sound file
GVCemoOutput emotions [500];
for (int step = 1; step <= 500; step ++) {
    GVCemoChannel_feed (
        channel,   // the only channel
        2000,
        buffer + (step-1) * 2000
    );
    GVCemoChannel_analyze (
        channel,   // the only channel
        3,   // probability
        & emotions [step - 1]
    );
}
```

4.3. Batch mode: two channels

```
GVCemoChannel incoming = GVCemoChannel_create (44100.0);
GVCemoChannel outgoing = GVCemoChannel_create (44100.0);
double buffer1 [1000000], buffer2 [1000000];
// fill the buffers with speech, perhaps from the left and right channel of a sound file
GVCemoOutput emotions1 [500], emotions2 [500];
for (int step = 1; step <= 500; step ++) {
    GVCemoChannel_feed (incoming, 2000, buffer1 + (step-1) * 2000);
    GVCemoChannel_feed (outgoing, 2000, buffer2 + (step-1) * 2000);
    GVCemoChannel_analyze (incoming, 3, & emotions1 [step - 1]);
    GVCemoChannel_analyze (outgoing, 3, & emotions2 [step - 1]);
}
```

4.4. Real-time mode

If you have a continuous stream of input samples, you can make use of the fact that 'collectingBuffer' is a ring buffer, i.e. 'totalNumberOfCollectedSamples' can be greater than 'collectingBuffer' and the last sample collected is found at collectingBuffer [totalNumberOfCollectedSamples % collectingBufferSize]. Make sure that the buffer is at least 5 seconds long, i.e. collectingBufferSize should be at least 220500.

In iOS, we need two different threads:

```
GVCemoChannel channel = GVCemoChannel_create (44100.0);
static NSLock *theLock = [[NSLock alloc] init];
static struct {
    double buffer [132300];   // a 3-second buffer in the recording thread
    int32_t numberOfSamplesCollected;   // starts at 0
} intermediary;   // apply theLock whenever you access any member of this structure
```

```
...

static OSStatus recordingCallback (   // set with AudioUnitSetProperty(...,
        // kAudioUnitProperty_SetRenderCallback, kAudioUnitScope_Input...)
    void                        *inRefCon,
    AudioUnitRenderActionFlags  *ioActionFlags,
    const AudioTimeStamp        *inTimeStamp,
    UInt32                      inBusNumber,
    UInt32                      inNumberFrames,
    AudioBufferList             *ioData)
{
    AudioUnit audioUnit = (AudioUnit) inRefCon;
    AudioUnitRender (audioUnit, ioActionFlags, inTimeStamp, 1, inNumberFrames, ioData);
    SInt32 *data = (SInt32 *) (ioData -> mBuffers [0]. mData); // 0 is the channel number
    [theLock lock];
    // we enter a section that is guarded by the lock because `intermediary` is accessed
    for (int i = 0; i < inNumberFrames; i ++) {
        float value = data [i] / (float) (1 << 24); // if the data is in fixed 8.24 format
        intermediary.buffer [intermediary.numberOfSamplesCollected + i] = value;
            // intermediary is guarded by the lock
    }
    intermediary.numberOfSamplesCollected += inNumberFrames;
    // we no longer need to access `intermediary`: leave the locked section immediately
    [theLock unlock];
    // perhaps set the contents of ioData to zero
    return err;
}

- (void) analysisTimer   // set with e.g. [NSTimer scheduledTimerWithTimeInterval:...]
{
    [theLock lock];   // start accessing intermediary
    int32_t numberOfSamples = intermediary.numberOfSamplesCollected;
    if (numberOfSamples > 0) {
        // copy part of intermediary to the main thread:
        GVCemoChannel_feed (channel, numberOfSamples, intermediary.buffer);
        intermediary.numberOfSamplesCollected = 0;   // restart the intermediary buffer
    }
    // stop accessing intermediary
    [theLock unlock];

    if (numberOfSamples == 0) return;

    GVCemoOutput emotions;
    GVCemoChannel_analyze (channel, 220500, 3, & emotions);
    if (emotions. numberOfValidFrames > 0) {
        // here should go some nice visualization of the happiness level
        printf ("%f\n", emotions. happyLevel);   // just a stand-in for a visualization
    }
}
```

Here we see that the locking time is minimized: samples are put into `intermediary` by the recording callback, and retrieved from `intermediary` by the analysis timer. In both cases, the section guarded by the lock does no more than copying samples from one buffer to another. This is why GVCemo makes a distinction between

`feed` (microseconds) and `analyze` (milliseconds); see also §3.2.4.

5. Python binding

The information in this chapter supplements the information given in §2.3.

If the language you are programming in is Python, the way to call the C function `GVCemo_compute()` is via the Python module `gvcemo`. This module uses `ctypes`, a package standardly included in Python that enables Python apps to call C functions.

The following two pages show the source code of the module `gvcemo`.

```
# gvcemo.py
# Copyright (C) 2015 Good Vibrations Company B.V.
# version 2015-09-06

import ctypes


class Output(ctypes.Structure):                    # In C: GVCemoOutput
    _fields_ = [                                   # Fields in C:
        ("num_valid_frames", ctypes.c_int),        #   int numberOfValidFrames
        ("num_frames_lost",  ctypes.c_int),        #   int numberOfFramesLost
        ("happy_level",      ctypes.c_double),     #   double happyLevel
        ("relaxed_level",    ctypes.c_double),     #   double relaxedLevel
        ("angry_level",      ctypes.c_double),     #   double angryLevel
        ("scared_level",     ctypes.c_double),     #   double scaredLevel
        ("bored_level",      ctypes.c_double)]     #   double boredLevel


_lib = None

def load(full_or_relative_path_to_GVCemo_library):
    global _lib

    # Get the names of the functions in the GVCemo shared object library:
    _lib = ctypes.CDLL(full_or_relative_path_to_GVCemo_library)

    # The shared library gave us the names but not the prototypes,
    # so we specify all the prototypes here (adapted from GVCemo.h):

    _lib.GVCemo_setWarningLevel.restype = None          # Return type in C: void
    _lib.GVCemo_setWarningLevel.argtypes = [            # Argument in C:
        ctypes.c_int]                                   #   int warningLevel

    _lib.GVCemoChannel_create.restype = ctypes.c_void_p # Return type in C: GVCemoChannel
    _lib.GVCemoChannel_create.argtypes = [              # Argument in C:
        ctypes.c_double]                                #   double samplingFrequency

    _lib.GVCemoChannel_feed.restype = None              # Return type in C: void
    _lib.GVCemoChannel_feed.argtypes = [                # Arguments in C:
        ctypes.c_void_p,                                #   GVCemoChannel channel
        ctypes.c_int,                                   #   int numberOfSamples
```

```
        ctypes.POINTER (ctypes.c_double)]               #   double samples []

    _lib.GVCemoChannel_analyze.restype = None           # Return type in C: void
    _lib.GVCemoChannel_analyze.argtypes = [             # Arguments in C:
        ctypes.c_void_p,                                #   GVCemoChannel channel
        ctypes.c_int,                                   #   int emotionFormat
        ctypes.POINTER (Output)]                        #   GVCemoOutput *output

    _lib.GVCemoChannel_reset.restype = None             # Return type in C: void
    _lib.GVCemoChannel_reset.argtypes = [               # Argument in C:
        ctypes.c_void_p]                                #   GVCemoChannel channel

    _lib.GVCemoChannel_delete.restype = None            # Return type in C: void
    _lib.GVCemoChannel_delete.argtypes = [              # Argument in C:
        ctypes.c_void_p]                                #   GVCemoChannel channel

    _lib.GVCemo_hasError.restype = ctypes.c_int         # Return type in C: int
    _lib.GVCemo_hasError.argtypes = []                  # No arguments in C

    _lib.GVCemo_getError.restype = ctypes.c_char_p      # Return type in C: const char *
    _lib.GVCemo_getError.argtypes = []                  # No arguments in C

    _lib.GVCemo_clearError.restype = None               # Return type in C: void
    _lib.GVCemo_clearError.argtypes = []                # No arguments in C


# Wrappings:

class Channel:

    def __init__(self, sampling_frequency):
        self._emo = _lib.GVCemoChannel_create(sampling_frequency)
        if _lib.GVCemo_hasError():
            raise RuntimeError(_lib.GVCemo_getError().decode("UTF-8"))

    def feed(self, number_of_samples, samples):
        _lib.GVCemoChannel_feed(self._emo, number_of_samples, samples)

    def analyze(self, emotion_format, emotions):
        _lib.GVCemoChannel_analyze(self._emo, emotion_format, emotions)

    def reset(self):
        _lib.GVCemoChannel_reset(self._emo)

    def delete(self):
        if not _lib is None:
            _lib.GVCemoChannel_delete(self._emo)


def DoubleArray(size):                          # For creating C double array arguments.
    return (ctypes.c_double * size)()
```

The module `gvcemo` translates all elements of `GVCemo.h` into Python: the C-struct `GVCemoOutput` has become

the class `gvcemo.Output`, and the C-function `GVCemoChannel_analyze()` has become the Python function `gvcemo.Channel.analyze()`.

You can call all wrapper functions (`create()` through `delete()`) with the arguments described in chapter 2. You can supply almost all arguments in the normal Python manner. The `output` argument to `analyze` has to have been created as an instance of class `gvcemo.Output`, and the `samples` argument to `feed` has to have been created as a `gvcemo.DoubleArray`, which works in much the same way as Python's built-in array of doubles.

5.1 Example: analyzing a sweep

```
# happy_sweep.py
# Computes the development of the "happiness" in a sine sweep.

import gvcemo

try:
    emo = None

    gvcemo.load("../lib/libGVCemo_mac.so")

    sampling_frequency = 44100.0
    emo = gvcemo.Channel(sampling_frequency)

    #
    # Create a rising sweep.
    #
    duration = 20.0   # Seconds
    num_samples = int(duration * sampling_frequency)
    samples = gvcemo.DoubleArray(num_samples)

    from math import sin, pi
    for i in range(0, num_samples):
        time = i / sampling_frequency
        # the local frequency is 100 + time * 10, i.e. it rises from 100 to 300 Hz
        samples[i] = sin(2 * pi * (100+time*5) * time)

    #
    # Make space for output.
    #
    output = gvcemo.Output()

    #
    # Compute and report the happiness level at multiple points in time. It should rise.
    #
    print("Happiness:   Valid frames:   Frames lost:")
    duration_of_slice = 2.0   # Seconds
    num_samples_per_slice = int(duration_of_slice * sampling_frequency)
    slice = gvcemo.DoubleArray(num_samples_per_slice)
    for islice in range(0, 10):
        slice[:] = samples[num_samples_per_slice * islice :
                           num_samples_per_slice * (islice+1)]
        emo.feed(num_samples_per_slice, slice)
```

```
        emo.analyze(4, output)   # Percentages
        print("%.3f" % output.happy_level,
            output.num_valid_frames, output.num_frames_lost)

finally:
    if not emo is None:
        emo.delete()   # Clean up
```

```
## Happiness:   Valid frames:   Frames lost:
## 2.463 170 0
## 9.522 172 0
## 26.548 172 0
## 51.573 173 0
## 73.544 172 0
## 86.825 172 0
## 93.529 172 0
## 96.740 173 0
## 98.297 172 0
## 99.076 172 0
```

This output was generated with Python 3.4. The script will also work with Python 2.7, although the output will be laid out slightly differently (because the `print` command works differently in both Python versions).

5.2 Example: analyzing the happiness in a WAV file

```
#!/usr/local/bin/python3

# happyWav_mac.py
#
# Purpose:
#   measures happiness values found in a recording of a human voice
#
# Call syntax:
#   python happyWav_mac.py full_or_relative_path_to_wav_file.wav
#
# Return value:
#   first column: a list of happiness percentages between 0 and 100;
#   second column: the number of frames analyzed to determine each happiness percentage
#
# Requirements:
#   the path to a dynamically linkable library that contains the GVC emotion detection algorithm;
#   this path follows here (it can be a relative or a full path name; modify it for your case):

import sys
sys.path.append("../../API")

import gvcemo
gvcemo.load("../../lib/libGVCemo_mac.so")

#
# Read the WAV file that is given on the command line.
```

```
#
import scipy.io.wavfile
(sampling_frequency, samples) = scipy.io.wavfile.read (sys.argv [1])
assert sampling_frequency == 44100.0
buffer_size = len(samples)
c_buffer = gvcemo.DoubleArray(buffer_size)

emo = gvcemo.Channel (sampling_frequency)

#
# Compute and report the happiness level at multiple points in time.
#
print("Happiness:   Valid frames:   Frames lost:")
for step in range (1, buffer_size // 20000):
    output = gvcemo.Output()

    #
    # Copy the samples, which are Python-floats, to C-doubles.
    #
    c_buffer [:] = samples [:]

    emo.feed(step * 20000, c_buffer)
    emo.analyze(4, output)
    print("%.3f" % output.happy_level, output.num_valid_frames, output.num_frames_lost)
```

6. Java binding

There are several ways to call GVCemo from your Java program, one of which is the Java Native Interface. On request we can describe here the interface file, and how to employ it.

7. R binding

There are several ways to call GVCemo from your R script, among which .C and .Call. On request we can describe here how this is done (for .C), or describe the interface file (for .Call), and how to employ it.

8. Licencing

You can purchase licences that allow you to use the GVCemo dynamic libraries for research and development (with e.g. Python or R), and to deploy the statically linked GVCemo object files in an iOS, Windows, MacOS X, or Linux app that you distribute on an app store or some other way.

For details, pricing, and special arrangements, consult your representative at GVC.