Heartbeat sounds

Group 8 - Synth 101

Federico Vescovi, CP:10935407, Matr n°:220651

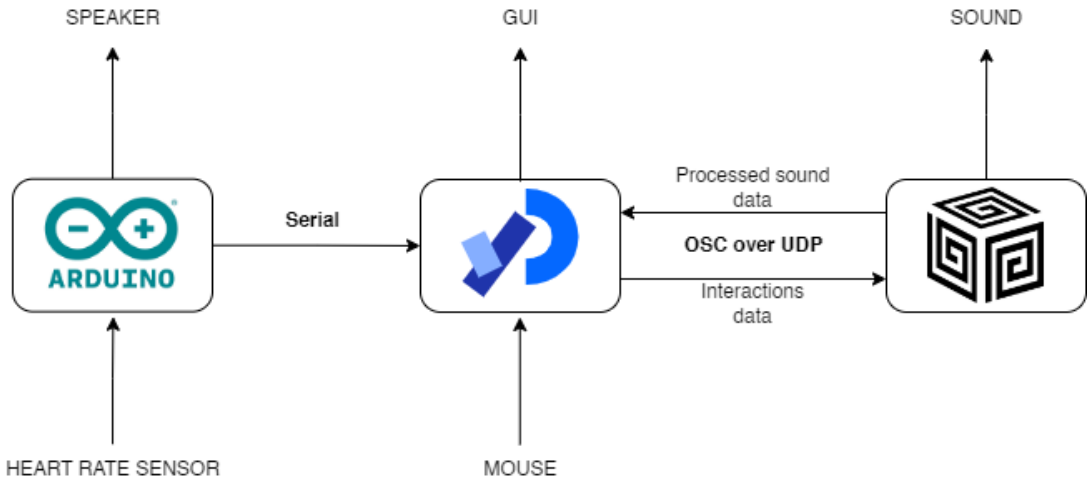
Pio Francesco Calogero, CP:10939480, Matr n°:226226

Marco Pelazza, CP:10746821, Matr n°:232343

Table of contents:

Introduction

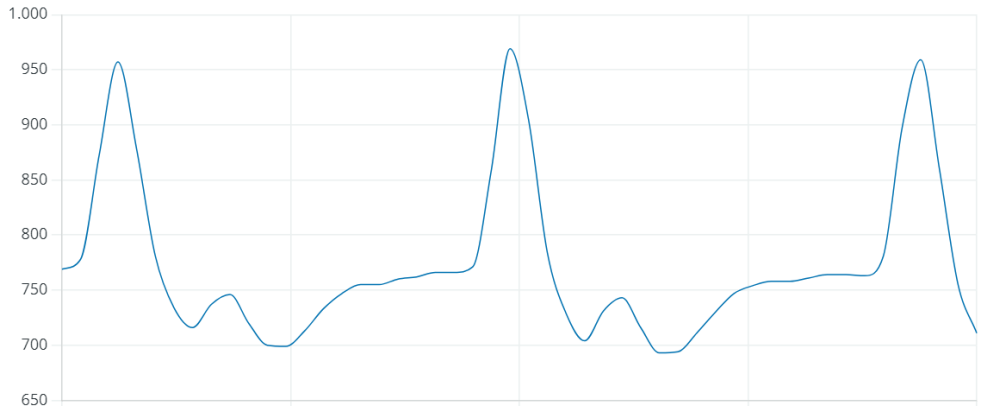
For this last homework the group has been asked to realize a full computer music system. We decided to implement an interactive musical environment that uses a heartbeat sensor and the data coming from the onscreen mouse position to modify several sound parameters. The sensor data in particular is processed by an Arduino board, in our case the MKR 1010, and afterwards sent through serial port to a PC.
For what concerns the realization of the GUI we decided to use Processing, in which we developed a code that has the job of handling all the mouse interactions and of managing most of the OSC message communication. More specifically, Processing receives the sensor data coming from Arduino and sends it along with mouseX and mouseY position data to Supercollider, where the actual sound generation happens. Finally, some important parameters are sent back from Supercollider to Processing so that they can be displayed.



Devices

For our project we decided to use two devices: a heartbeat sensor, set in its analog mode, and a speaker, in order to simulate a metronome.

1. **DFRobot Heart rate monitor sensor for Arduino:** This sensor has both an analog and a digital mode. In the beginning we did some testing with the second one and we quickly found out that it is not very accurate, plus it has an extremely low data flow, which is a big problem for a live environment like ours. For these reasons we decided to rely on the analog mode of the sensor, which simply makes the sensor act as a blood pressure level detector instead. We plugged the sensor in the A2 pin of our board and supplied a voltage of 3.3V that comes from the VCC pin. The range of values that quantize the blood pressure spans between 0 to 1023, by tracking the single values it is possible to plot the signal as in the graph represented below.
2. **DFRobot Digital speaker module:** we programmed this small speaker to emit a short sound for every heart pulse detected by the sensor. The speaker works at 0.5W of maximum power and we plugged it in the digital pin 8 providing it a 5V tension.



Arduino

In Heartbeat sounds, Arduino represents the foundations on top of which the whole system is built. We used the Arduino board both to collect data from the heart rate sensor and to begin the first part of their elaboration: the current BPM value is in fact computed directly on the board and then passed to the PC through the serial port. In the setup() function we initialized the serial, at a bit rate of 9600 baud, we also implemented the generation of a startup sound to identify when the board is turned on, which involves the reproduction of four simple notes using the Arduino built-in function tone(). The loop function starts reading the current value from the pin A2, that is the one that receives the values from the heart rate sensor. The incoming values are then checked since
we want to make sure that the user positioned the sensor neither too close, nor too far from its finger; if they are not in the [551, 998] interval the string “NVV” is sent to the serial, which stands for “Not Valid Value”. In the other case, the actual values are considered good enough and we proceed to send a serial message with the following structure:

```
String(val) + String(currentBPM) + "E"
```

Where “val” is the current value returned from the sensor and “currentBPM” is the heart rate and the “E” character is there only to identify the end of the message.

As it's possible to see in the sensor output graph above, the signal has periodical peaks. These peaks correspond to the systolic phase of the cardiac cycle, which is the one where the blood pressure is higher. This characteristic was exploited to compute the BPM. After some test we understood that graph values higher than 780/800 are reached only nearby a peak, but to add another layer of security in pulse detection we decided to implement also the difference based function: `bool isPeak (int last, int current)`
This function evaluates the difference between two samples received from the heart rate sensor, if it is higher than a certain threshold a boolean true is returned. The samples we decided to consider are the current real-time one and the 30th previous sample. This number has been chosen by considering the amount of samples present in the ascending section of the peak. The current heart rate is then calculated by the function `computeBPM(periods)` that returns the BPM by taking into consideration the time intervals between ten peaks. If this value is in the [40, 150] interval it is considered valid and written in the serial, if it is not, the number 33 is sent instead, which makes no sense for a person's heart. When receiving this value it means either that there weren't enough values to compute the BPM or that the BPM computation gave a not realistic result.
Additionally, we had to delay each iteration of the loop function in order to not overload the serial port.Finally, we implemented a feature that makes the speaker reproduce a note each time the high pressure state is reached to remind the user a metronome feeling driven by the beat of his own heart.

Processing

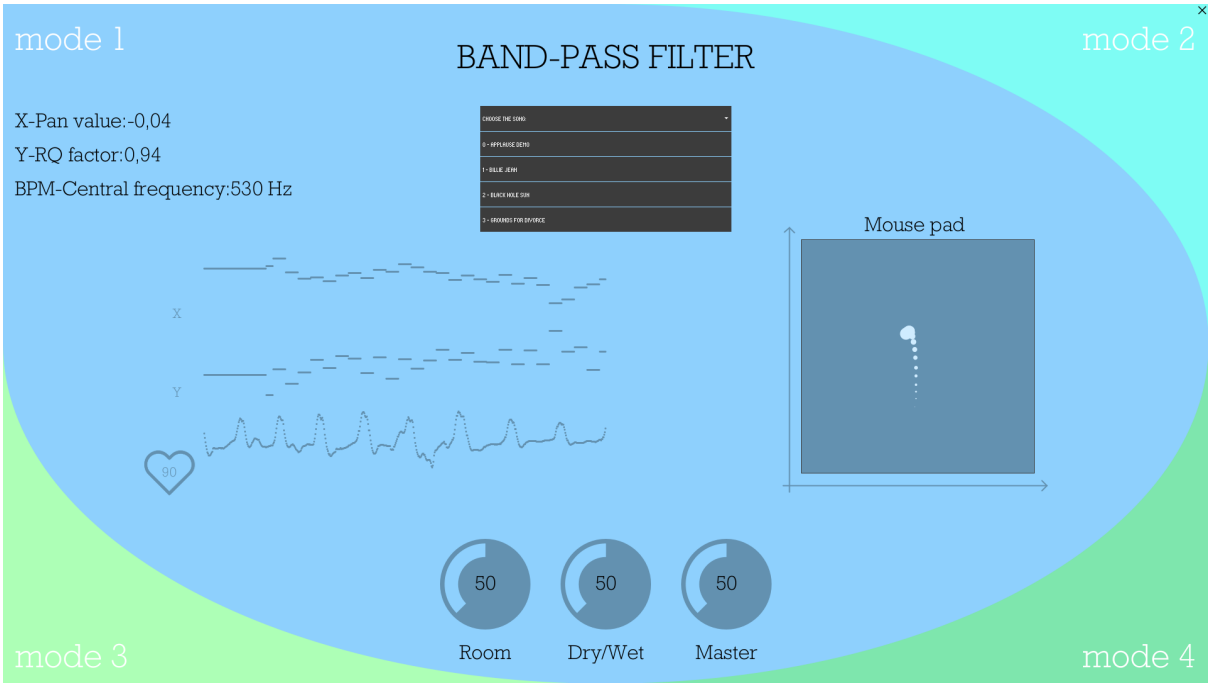
For what regards the Graphical User Interface of our project, we decided to make use of the software "Processing". The user can experiment with 4 possible different modes, each one of which has a different sound implementation and different uses of the three inputs (MouseX, MouseY and bpm value). We placed the four mode selectors on the corners of the interface, after clicking one of these the onscreen text gets refreshed so that it consistently reflects the audio parameters controlled in SuperCollider.
In Processing we also display the current MouseX, MouseY and Heart pressure value in three separate graphs shown in the middle of the screen. In every mode it is implemented a mousepad to control the X and Y values of the synths and a handy knob for the control of the Master volume. Furthermore, in mode 2 and 4 there is an extra knob that controls the type of wave utilized to synthetize the sounds (sinusoidal, sawtooth, etc.). On the other hand, in mode 1 and 3 there are other additional knobs, one for the reverb room and one for the reverb dry/wet, moreover a drop-down menu for the selection of the song has been added to the interface.
Other two very important code sections of our Processing file are the ones related to the serial and OSC. For what regards the information written by the Arduino, we used the Serial class of Processing to make the reading, and then we decrypted the message by decomposing the structure as we discussed before.
For what concerns the OSC messages instead, it's important to say that there are only two messages that pass through the entire system, one is /DataForSupercollider, and the other is /DataForProcessing.
The first one sends the following information to SuperCollider:

1. Currently selected mode number
2. MouseX, mapped from 0 to 1

3. MouseY, mapped from 0 to 1
4. Current bpm
5. Value of the currentSong selected in the drop-down menu
6. Value of the knobRoom knob, mapped from 0 to 1
7. Value of the knobDryWet knob, mapped from 0 to 1
8. Value of the knobMaster knob, mapped from 0 to 0.5 (we reduced the sound in order for it to be more controllable).
9. Value of the knobWave knob, which can have value: 1,2,3,4.

The second one sends the following information to Processing, which are meant to be displayed in the GUI:

1. The value of the parameter modulated by mouseX
2. The value of the parameter modulated by mouseY
3. The value of the parameter modulated by the bpm



In the next section we will see in detail how these OSC messages are used.

SuperCollider

The sound synthesis of our project is entirely made in SuperCollider through the use of 5 SynthDefs, 4 of these implement the Heartbeat sounds modes, while the last one realizes a delay effect.
Since we wanted the sounds to be very different with respect to each other, we splitted the four modes into two groups: the left modes (1 and 3) and the right modes (2 and 4).
The left modes run by reading some songs on the ~songs array of buffers.

```
(
~songs=Array.new;
~folder=PathName.new(thisProcess.nowExecutingPath.dirname+/+"songs")
)
(
~folder.entries.do({
    arg path;
    ~songs=~songs.add(Buffer.read(s,path.fullPath));
});
)
```

This way, when we want to read a song, we can simply call *~songs[songSelection],* where *songSelection* is an integer going from 0 to 10 representing the song number inside the "***songs***" folder of the project. On the other hand, mode 2 and 4 instead run by generating sounds through different SuperCollider Synths.

Receiving OSC messages

Now we're going to address how the OSC messages that we anticipated in the previous section are handled in SuperCollider. To understand the following lines we have to specify that we set the receiving OSC address to be *recAddr=NetAddr("127.0.0.1",57120);* and the sending address to be *sendAddr=NetAddr("127.0.0.1",12000);*. This way we are receiving local messages from port *57120,* and sending other local messages to port *12000*.
With that out of the way, the first OSC message our program runs into is the message /DataForSupercollider coming from Processing at port 57120. In order to save its information correctly in suitable variables, we created an OSCdef called *OSCreceiver*, where we initiated the variable *msg*. The received message is saved exactly in this variable, therefore we can simply access its information by doing as here below:

```
OSCdef('OSCreceiver',

    {
        arg msg;//addr= recAddr;
        ~mode= msg[1];

        x = msg[2];
        y = msg[3];
        bpm = (msg[4]).asInteger;

        songSelection =msg[5];

        ~room=msg[6];
        ~reverb =msg[7];
        ~master =msg[8];
        ~wave = (msg[9]).asInteger;
```

Once we have our data saved in our SuperCollider variables we can define our SynthDef more easily. Please keep these variables in mind when reading the following pages as we will refer to them from time to time.

Sending OSC messages

In order to show the correct information on the interface, SuperCollider has to send the data that modulates the various Synths to Processing. We do so by sending a message after

each time a Synth is created. For instance, for the first Synth we execute the following line of code: *sendAddr.sendMsg("/DataForProcessing", x, rq, freq.asFloat);*
which sends the */DataForProcessing* message to Processing, containing the values of *x*, *rq* and *freq*, all in a float representation. Processing then receives the message and saves each value on its own variables in a similar fashion to what mentioned above for SuperCollider and plots everything on the screen. Now that the OSC communication part has been dealt with, we will go over the various effect implementations.

Mode 1 - Band-Pass filter

When ~mode=1 we enter the first mode of our program, which is realized thanks to the use of the \effect1 SynthDef. This mode basically consists in a band-pass filter modulated in different manners by the following parameters:

x : modifies the pan of the signal, whose range is [-1, 1],
y : sets the range of $rq = \frac{bandwidth}{freq}$,
bpm : controls the central frequency of the bandpass filter, the relationship between *bpm* and such frequency is of the exponential type.

The first signal *sig* in the SynthDef corresponds to PlayBuf (a sample playback oscillator) declared as follows:

```
sig = PlayBuf.ar(numChannels:2, bufnum: ~songs[songSelection].bufnum, rate: 1, doneAction:0);
```

in which bufnum is the index of one of the eleven buffers contained inside ~songs, specifically the one corresponding to the currently selected song.
In order to realize the band-pass filter we used the class BPF.ar, this is where both *bpm* and y come into play, modifying respectively the frequency and the coefficient *rq* of the bandpass filter. The final output signal of the SynthDef takes into consideration the values of *x*, which is correlated to the pan, and of the general knobs *Room*, *Dry/Wet* and *Master*. It's relevant to notice that the parameters of our Synth are constantly refreshed thanks to the use of the Routine ~routine01 that continuously sets the updated synth arguments.

Mode 2 - Random notes

This effect deals with random generation of MIDI notes in a given interval, here the user input controls the following features:

x : modifies the pan of the signal, whose range is [-1, 1],
y : sets the value of *variation,* which determines the amplitude of the interval in which MIDI notes can be generated,
bpm : determines the value of *f*, which is the central frequency of such an interval.

In the SynthDef of this effect, we used an envelope that establishes the duration of each note. In other words, we defined EnvGen.kr as follows:

```
env = EnvGen.kr(Env.perc(0.01, sustain, 0.2), doneAction: Done.freeSelf);
```

obtaining an envelope with a percussive shape, suitable for the reproduction of a single note. In this mode, the user can also choose different synthesizers, in order: SinOsc.ar, LFSaw.ar, Pulse.ar and LFTri.ar. These respectively correspond to the following numbers of the knobWave (1, 2, 3, 4). These various oscillators take *p* as frequency and *env* as the multiplying factor. One final detail that still needs to be discussed is how we decided to generate the random notes, and that is shown in the lines of code here below:

```
var variation = (y*20).trunc.asInteger;
f =(bpm*(108/149)).trunc.asInteger;
 rand = rrand(f-variation, f+variation);
p = rand.midicps;
```

The key points here are the following:

- *variation* controls how wide is the range of MIDI notes from which we can generate sounds. The maximum range is ±20 MIDI notes.
- The MIDI note *f* that represents the central frequency of the range can go from 29 to 108.
- *p* converts the notes from MIDI notation to their respective frequency value in Hz.

In order for the mode to work, we implemented two routines, one to get the next note that will be played and one to actually play the note. Inside each routine there is a *loop*, in which we inserted the command **“** *0.1.wait;* **”** in order to make two consecutive notes be delayed by 0.1 seconds.

Mode 3 - Song accelerator

The third effect implements an acceleration (or deceleration) of the current speed of a song, in particular:

x : modifies the pan of the signal, whose range is [-1, 1],
y : controls the song speed, its range is [0,2];
bpm : determines the delay time applied to the song. Ranges from 0 to 1.

More specifically, the “speed up” effect is given by the number of the variable *rate* in the following line of code: *sigOut=PlayBuf.ar(2, ~songs[songSelection].bufnum, rate, loop:0)*, which basically controls the speed rate at which we evaluate the buffer containing the selected song. The delay is instead made possible thanks to an independent Synth with respect to the one of \effect3, called \myDelay. In this SynthDef we basically take what is currently being played in the output channels and delay it by the variable amount of time *delaytime.* The line of code that does this is the following: *DelayN.ar(input, 2,delaytime);.* After doing so we can send the delayed signal to the output channels. It’s relevant to mention that in order for the two Synths to work in the correct order we had to define two SuperCollider groups and specify their order.

Mode 4 - Arpeggiator

The fourth effect is an arpeggiator based on an infinite repetition of four notes (a major triad plus the octave). The SynthDef is regulated by the following parameters:

x : modifies the pan of the signal, whose range is [-1, 1],
y : sets the value of *f* that corresponds to the MIDI note of the tonic,
bpm : sets the duration of the arpeggio.

In the SynthDef of this effect, we used a percussive envelope that establishes the duration of each note; such envelope is completely identical to the one described in the second effect. As in mode 2, the user can choose between the aforementioned synthesizers to which such envelope is applied. The sequence of midinotes are set according to the following assignments, which are repeated every time we end the fourth note of the arpeggio:

```
f = 127-(y*117).trunc.asInteger;
l = [f, f+3, f+7, f+12];
a = Pseq(l, inf).asStream;
```

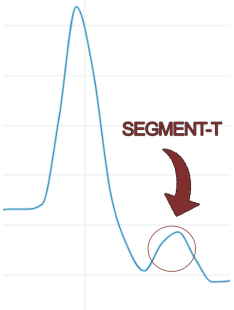
This assignment is instead repeated after every note:

```
freq = a.next.midicps;
```

Where *freq* is the frequency of the note that will be heard by the user.

Known issues

- When the variation of the pressure during the peak is small, in the specific lower than 80, the peak is not detected and so the BPM is not calculated. We decided to not decrease the thresholds for peak detection because, in that case, the segment-T would be considered as another maximum, causing an incorrect computing of BPM. This problem is not actually common but it typically occurs when the sensor is not positioned in a correct way.



- Having the processing GUI in background will cause a delay of the serial reading. If that happens, when the user returns to our program, the signal that represents the heart trend is going to be extremely delayed because the accumulated queue has to be emptied.
- Since the heartbeat sensor is pretty delicate, it is substantially difficult to use the device after/during physical activities.