Report homework 3

Group 8 - Synth 101

## 

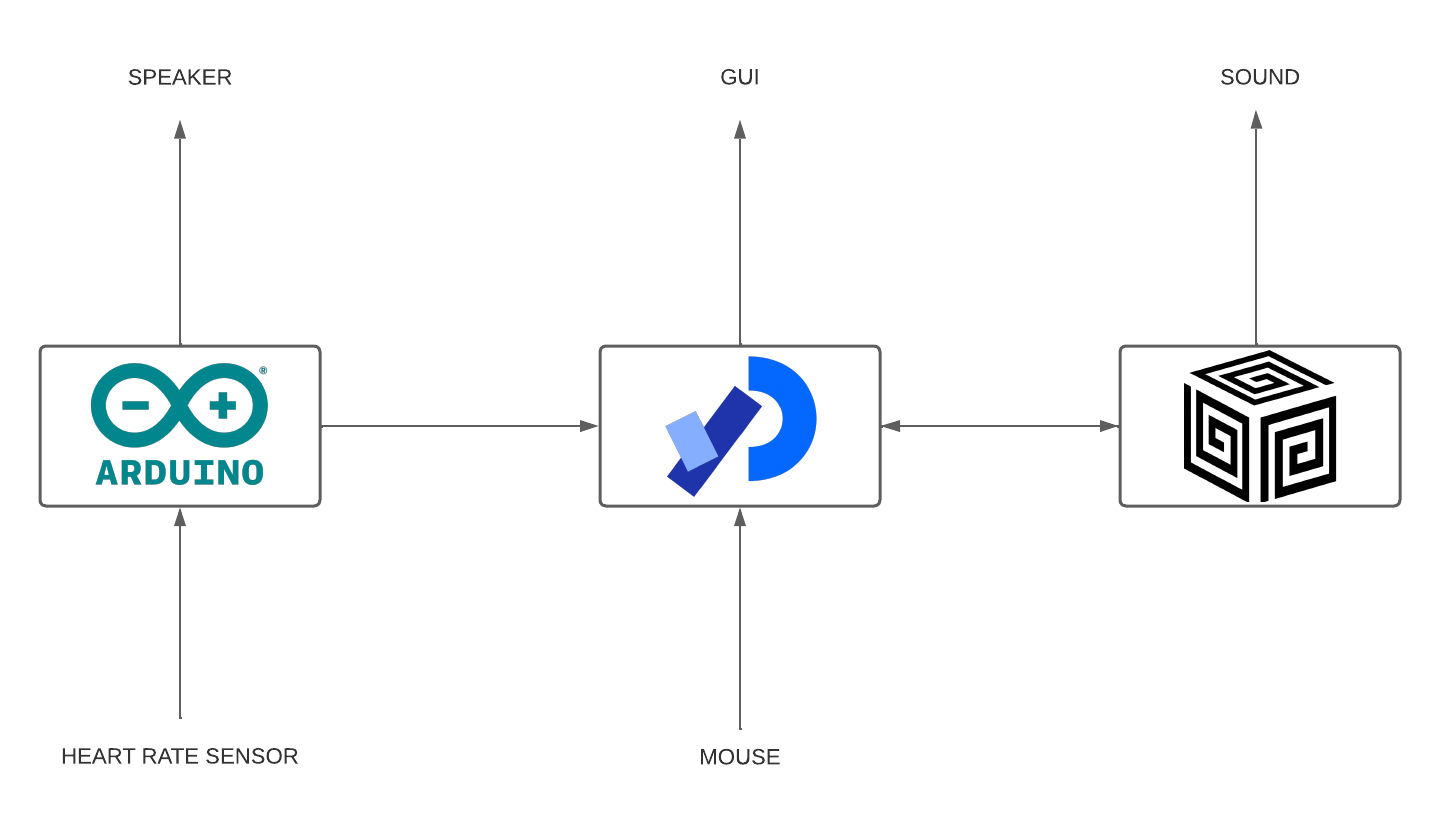
## Table of contents:

* Introduction
* Sensors
* Arduino
* Processing
* Supercollider

## Introduction

For the last homework the group has been asked to realize a full computer music system. We decided to implement an environment where the interactions are mostly dependent on data coming from an heartbeat sensor and the onscreen mouse position. The sensor data in particular are processed by an Arduino board, in our case the MKR 1010, and afterwards sent through a serial port to a PC.

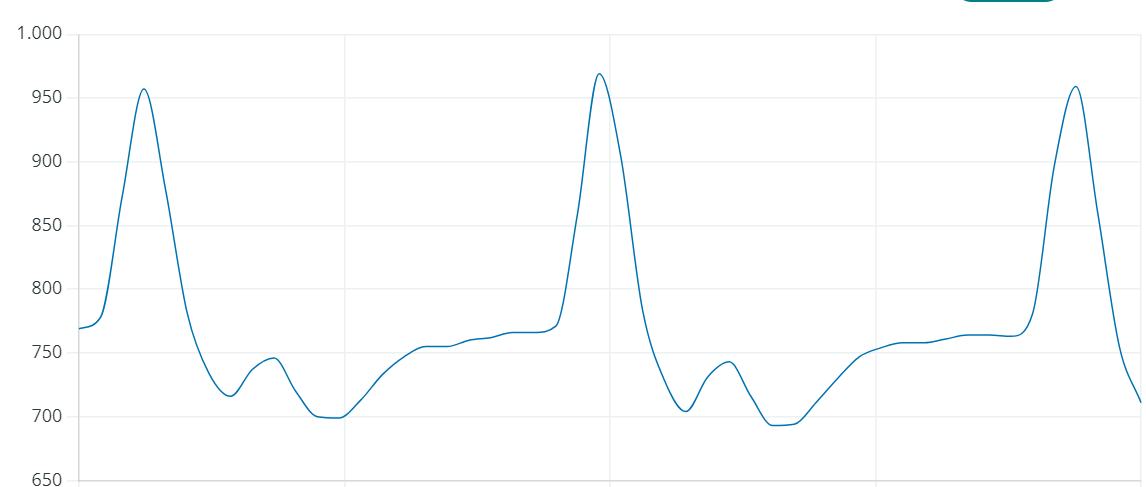
For what concerns the realization of the GUI we instead decided to use Processing. In our code, this is both where all the mouse interactions are implemented and where most of the OSC messages get sent/received. In detail Processing receives the sensor data coming from Arduino and sends it along with mouseX and mouseY position data to Supercollider, where the actual sound generation happens.



## Devices

For our project we decided to use two devices: a heartbeat sensor, used in its analog mode, and a speaker, used to reproduce sound.

1. **Heartbeat sensor:** We used the DFRobot “Heart rate monitor sensor for -Arduino”. Even if a digital mode is implemented from the factory, we decided to not use it, because it’s not accurate and the data flow is extremely low. We use the sensor instead with his analog mode, we plugged it in the A2 pin of our board and we supplied a voltage of 3.3V that comes from the VCC pin. It is basically a pressure sensor that when applied, for example, on your finger is sensitive to the blood pressure change and reports it . The range of values provided are from 0 to 1023, so they could be mapped as in the graph represented in image 1.
2. **Speaker:** We used the DFRobot “Digital speaker module”, a small speaker that works at 0.5W of maximum power. We plugged it in the digital pin 8, and provided it with a 5V tension.



## Arduino

We used the board both to collect data from the heart rate sensor and to make part of their elaboration . In fact the current BPM value is computed directly on the board and then passed to the PC through the serial port. In the setup() function we initialized the serial, with a bit rate of 9600 baud. We also implemented the generation of a start sound, to identify when the board is turned on, that reproduces four simple notes using the Arduino built in function tone(). The loop function starts reading the current value of the pin A2, the one that receives the values from the heart rate sensor. These values are then checked, if it’s in the [551, 998] interval we consider it as good as to be written in the serial and sent to the PC. We introduced this control because if the values are lower the sensor is not put in contact with the body of the user, and if they are bigger probably the sensor is moving. We came across these results testing the sensor for some time.

After that if the value is ok, we print it in the serial followed by the current value of BPM. The heart rate is computed afterwards in the code but we initialize it with the value 33. This value could be interpreted as a code to identify a moment when there aren’t enough values to compute the BPM or its computing gives a not realistic result.

If the first check has a negative result the string “NVV” is then sent to the serial, this could another time be interpreted as a code to communicate that the values returned from the sensor aren’t valid.

The message sent, if the value pass the control to, the serial has the following structure:

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

Where val is the current value returned from the sensor, currentBPM is the heart rate. We decided to print the “E” character to identify the end of the message.

After some tests we understood that if val is greater than 780/800 (obviously depending on the person), we are in a high pressure moment that corresponds to a beat of the heart. So the board has to basically count the amount of time that passes before reaching another peak. We chose to compute the BPMs every ten peaks, to have more stable values of them, so we initialize a vector in this way: int periods[10]

This vector is useful to memorize the values of the different periods between the peaks, if it has reached ten members we execute the function int computeBPM(int arr[])

This function is the one that, as its name says, computes the current value of BPM. It is quite simple, it basically extract each value of the ten instantaneous rate, sum them and then divide the sum by ten, obtaining a mean value over the ten periods.

The value returned by computeBPM() is then checked and if it’s not in the range [61, 129], we put currentBPM equal to 33. When a peak is reached a high note is also played by the speaker, we use tone() another time, to recreate a metronome based on the heart beat.

To avoid the identification of false peaks and so the playing of a note in a wrong instant of time, we used a flag called notePlayed.

The whole cycle is executed once every 50 milliseconds, we set this delay to avoid the overloading of the serial.

## 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 on the middle of the screen. Furthermore, we made use of three knobs, one for the selection of the song, one for the reverb and one for the Master volume.

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 encrypted the message by decomposing the structure that we discussed before.

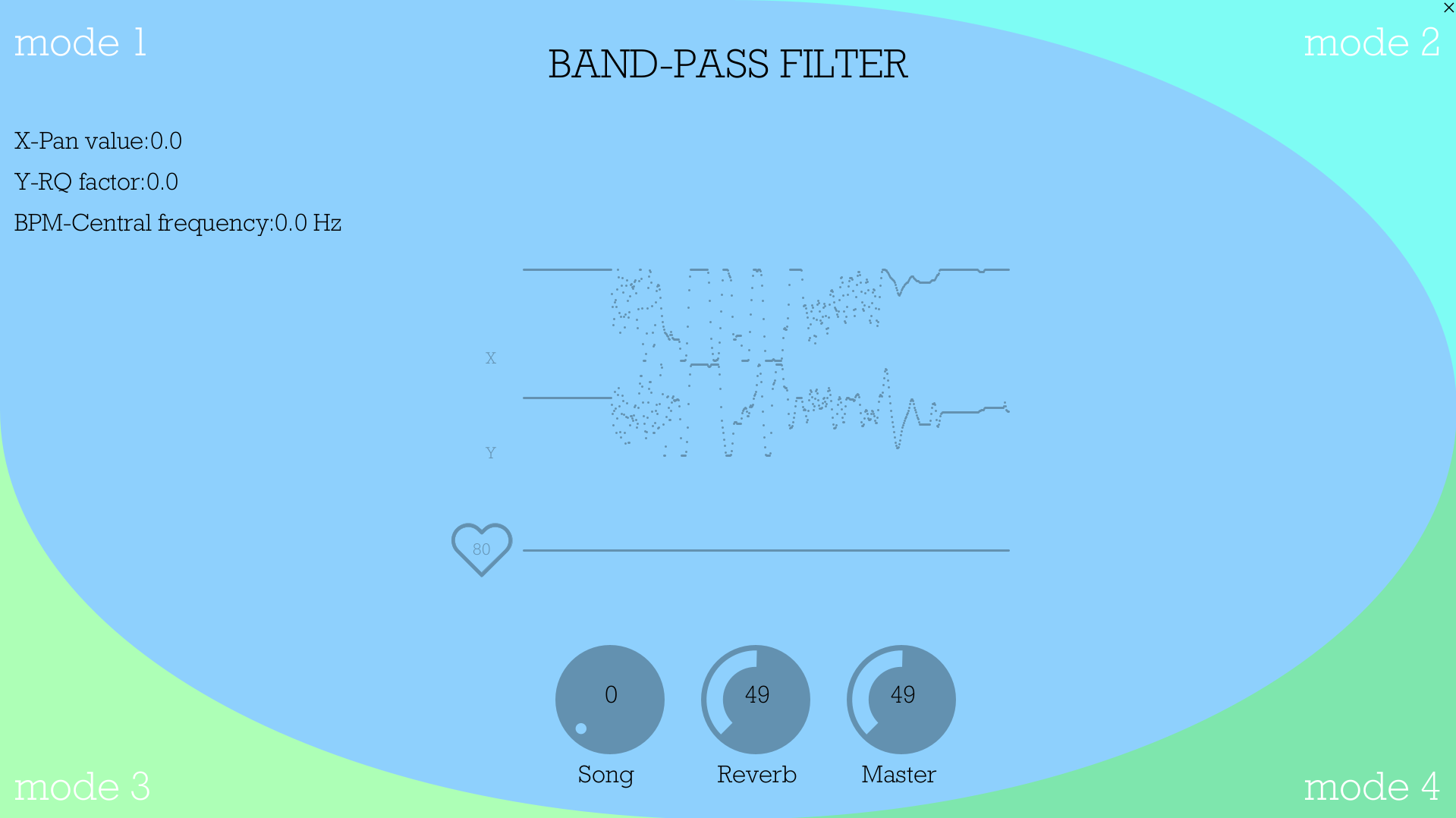
For what concerns the OSC messages instead it’s important to say that there are only two messages that pass through the entire system, one is /DataForSupercollider, and the other is /DataForProcessing.

The first one sends the following informations 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 knobSongSelection knob,
6. Value of the knobReverb knob, mapped from 0 to 0.5
7. Value of the knobMaster knob, mapped from 0 to 0.5 (we reduced the sound in order for it to be more controllable).

The second one sends the following informations 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



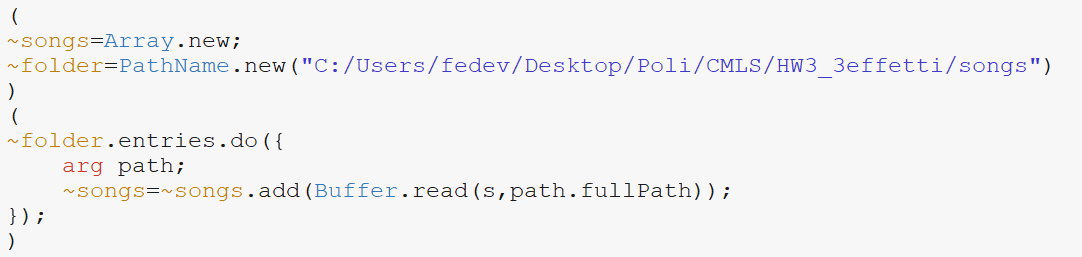
In the next section we will see in detail how these OSC messages are used.

## SuperCollider

The sound synthesis of our project is entirely made in Supercollider through the use of 5 synthDefs, 4 for these implement the HeartSoundGenerator modes and the last one for a delay effect implementation.

Since we wanted to make the sounds very different with respect to each other, we splitted the four modes in two: 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.

****

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. On the other hand, mode 2 and 4 instead run by generating sounds through different sinOscs synthetized in supercollider.

### 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 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 initiate the variable *msg*. The received message is saved exactly in this variable, therefore we can simply access its informations by doing as here below:

****

Once we have our data saved in manageable variables we can define our SynthDef more easily.

### 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 after : *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. In the following sections we will go over the various effect implementations.

### First Effect

In order to select the desired effect we can inspect the value of ~mode. In this case, once the user has set ~mode = 1, the first effect will be chosen. The definition of the \effect1 is done by exploiting the class SynthDef in SC and basically consists in a band-pass filter in which the following parameters modulate different arguments:

*x* : modifies the pan of the signal, whose range is [-1, 1],

*y* : sets the range of **,**

*bpm* : is the parameter through which the function “linexp” maps an exponential range of values by determining the value of *freq* (central frequency in Hz)

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 corresponds to the index of the buffer used by ~songs and “songSelection” is the value through which the user can set the song of interest.

In order to realize the band-pass filter, we use the class “BPF.ar” and pass as arguments our input signal *sig*, the central frequency *freq* and *rq* so that the bandwidth is computed with respect to *freq*. Then we send this final signal to “Out.ar”, set the multichannels output and finally we apply the pan by using “LinPan2.ar”.

It’s relevant to notice that the aforementioned parameters (*freq, rq*) are located in a “Routine” in order to be refreshed according to the current values of *x*, *y* and *bpm* received by Processing.

### Second Effect

This effect deals with the random generation of midi notes by modifying the following parameters:

*x* : modifies the pan of the signal, whose range is [-1, 1],

*y* : sets the value of *variation* that then determines whether the range of the midi note numbers is increased or decreased,

*bpm* : is the parameter that determines the value of *f* (frequency in Hz)

In the SynthDef of this effect, we use an envelope that helps us to establish the duration of each note. Therefore we define “EnvGen.kr” as follows:

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

basically, we pass an “Env” instance with a percussive shape whose arguments are the duration of the attack time equals “0.01”s, the duration of the release time equals to the arg “sustain” (0.05 s) and the peak level of the envelope set as “0.2”s.

The main signal is *sig* that is a “SinOsc.ar”, whose frequency is *p* and the multiplying factor is *env*. Then we send this last signal (*sig*) to “Out.ar”, set a mono channel and finally we apply the pan by using “LinPan2.ar”.

The random generation is created according to the following assignments:

var *variation* = (y\*20).trunc.asInteger;

*f*=(bpm).trunc.asInteger;

*r* = rrand(f-variation, f+variation);

*p* = r.midicps;

the variable *r* establishes the lower and the upper limit of the calculated random numbers. Then *p* converts these midinotes into cycles per seconds (Hz).

### Third Effect

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

*x* : modifies the pan of the signal, whose range is [-1, 1],

*y* : speed rate at which we evaluate the buffer containing the selected song, this is what gives the ”speed up” effect. Ranges from 0 to 2, 1 is the normal song speed,

*bpm* : determines the delay time applied to the song. Ranges from 0 to 1.

More specifically, the “speed up” effect is gave by the number of the variable *rate* in following line of code: *sigOut=PlayBuf.ar(2, ~songs[songSelection].bufnum, rate, loop:0)*, which speeds up or slows down the song by evaluating the samples of the buffer with higher or lower speed.

The delay is instead made thanks to an independent Synth with respect to the one of \effect3, called \myDelay. In this SynthDef we basically take what is currently 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 synth to work in the correct order we had to define and specify the order of two supercollider groups.

### Fourth Effect

The fourth effect is arpeggiator, which is based on an infinite sequential repetition of a list of notes. 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 tonic midinote,

*bpm* : is the parameter that sets the duration of the arpeggiator.

In the SynthDef of this effect, we used an envelope that establishes the duration of each note. After that, we defined “EnvGen.kr” as previously described in the second effect with the same arguments. The main signal is *sig* that is an instance of “SinOsc.ar”, whose frequency is *freq* and whose multiplying factor is *env*. The sequence of midinotes are set according to the following assignments:

*f* = 127-*(y*\*117).trunc.asInteger;

*l* = [*f*, *f*+3, *f*+7, *f*+12];

*a* = Pseq(*l*, inf).asStream;

*freq* = *a*.next.midicps;

The list *l* depends on yand composes the sequence. This is then infinitely repeated by using “Pseq” and lastly *freq* scans all the values of *l*. In conclusion all of the aforementioned variables are continuously refreshed through a “Routine” in loop.