SYNTHBERRY PI: AN AUTONOMOUS SYNTHESIZER BASED ON RASPBERRY PI

Costantino Rizzuti

Artis Lab
Cosenza, Italy
costantinorizzuti@gmail.com

Fabrizio Rizzuti

Artis Lab
Cosenza, Italy
fabrizio.rizzuti@gmail.com

ABSTRACT

SynthBerry Pi is the first prototype of an autonomous synthesizer based on PDSynth. PDSynth is a toolkit for creating programmable digital synthesizers made using the Pure Data visual development environment. To run PDSynth synthesis architectures a Raspberry Pi mini computer was used. Eight slide potentiometers are connected to the mini computer to create a control surface that makes it possible to control PDSynth's architectures. SynthBerry Pi is, therefore, a compact standalone synthesizer capable of creating sounds by using Pure Data patches.

1. INTRODUCTION

In about the last twenty years, the miniaturization of computing systems and the growing diffusion of open software and hardware technologies allowed artists and designers to access technologies until then only available for technicians and engineers working in large university or business research centers. As Noble [1] mentions, all of this has created absolutely new and never seen conditions, making possible the emergence of new fields of research in art, design and also music such as: Physical Computing and Interaction Design. In fact, before that the idea of artists or designers writing code or designing hardware was almost unheard of. Today, not only it has become commonplace, but it has become an important arena of expression and exploration. Nowadays, this deep bound between technology and artistic creation is become a vital and vibrant phenomena that shapes both art and technology.

Even in computer music the growth of this technologies lead to interesting consequences like the positive convergences with already existing trends such as the practice of self-constructing synthesizers, the development of new musical interface and the building of experimental electronic musical instruments. These practices defined as *analog synthesizer do it yourself*, in its abbreviated form: *synth DIY* [2], aimed at the realization of electronic musical instruments, have had a great diffusion in recent years especially in relation to the increasing use of modular eurorack synthesizers.

This work presents *SynthBerry Pi*: a prototype of an autonomous synthesizer based on Raspberry Pi mini computer. The prototype uses Pure Data patches to generate and process sounds. The collection of patches used for this project is called *PDSynth*: a toolkit for creating programmable digital synthesizers. The Raspberry Pi mini computer was used to run PDSynth synthesis architectures in order to create a compact standalone synthesizer. SynthBerry Pi is equipped with an hardware control interface consisting of eight slide potentiometers that allow to change the parameters of the Pd patches.

Raspberry Pi[1] is a well known mini computer. The project is based on a Broadcom *system-on-a-chip* (BCM2836 for the Raspberry Pi 2, or BCM2837 for Raspberry Pi 3 and BCM 2711 for the latest Raspberry Pi 4 Model B), which incorporates an ARM processor, a VideoCore IV GPU and RAM memory (from 512 Megabytes, to 1 Gigabyte, up to 4GB for the latest version). The boards do not have neither hard disk nor a solid state memory unit, relying instead on an SD card for the boot and for the management of the non-volatile memory.

In the last years many audio projects have been realized around the Raspberry Pi platform [3, 4, 5, 6]. Moreover, Eurorack modules, such as the Terminal Tedium project and Nebulae 2 from Qu-Bit Electronix, use Raspberry Pi to create reprogrammable modules that can be used to implement audio processes developed through languages and development environments for audio (from C and C++ as programming languages, up to to Pure Data, SuperCollider and CSound as languages dedicated to audio).

2. PDSYNTH

PDSynth is a toolkit for creating programmable digital synthesizers.[2] The name derives from the acronym of the sentence: *Programmable Digital Synthesizer*. But also the acronym *PDSynth* allows to indicate a synth made with Pure Data (Pd) [7]: the well known visual development environment for multimedia applications used to implement the toolkit.

The development of this project started in the Autumn 2015 from an initial idea to create a series of easily interfaceable Pd patches capable of simulating the behavior of the essential modules of an analog synthesizer. The *PDSynth* modules implement different sound generation and processing systems and are all controllable via the *Open Sound Control (OSC)*[3] protocol. Users can easily create and interconnect the modules together to build high-level architecture for real-time sound synthesis and processing. The OSC protocol [8, 9] was choosen because it is becomed, along the years, a standard format for sharing data related to musical performance (parameters, sequences of notes, gestures) between musical instruments (mainly synthesizers and electronic instruments), calculators and other multimedia devices. This protocol, from the late 1990s, is becomed a valid alternative to MIDI, especially because it is open, flexible and extendable.

Open Sound Control was choosen because it allows to easily create a reliable and robust communication system among the various modules inside Pd allowing, also, a simple exchange of network messages to and from the outside. In fact, the OSC messages can be easily managed through the native message system provided by Pure Data. All this simplifies the creation of the control systems of the modules and allows to control the synthesizers through external

[1] For more information refer to the Raspberry Pi Foundation website: https://www.raspberrypi.org/.

[2] The PDSynth toolkit can be downloaded from Artis Lab website: https://www.artislab.it.

[3] For further information, refer to the project's official website: http://opensoundcontrol.org.

controllers and control surfaces. Moreover, the OSC protocol allows to have both an higher data resolution and greater parameter space than what is offered by the MIDI protocol.

A library of external objects was used to implement the control functions through the OSC protocol. The library provides useful objects only to realize the management of the OSC messages inside Pure Data, so for the transfer of OSC packets through the network, a second library, called *IEMnet*, was used. In particular, it is possible to use the `udpreceive` object to implement within Pd a server listening on a given port for receiving OSC messages.

The processing of messages received from the server can then be carried out using the objects provided by the OSC library. The `unpackOSC` object is useful for converting OSC packages, made up of binary data, into messages compatible with the Pure Data internal messaging system. Then the `pipelist` object is inserted to obtain a temporally coherent message scan in the case in which messages with a given *timestamps* are received. Finally, the OSC library provides an object, called `routeOSC`, which allows the addressing of messages according to a hierarchical structure defined by the address space. The arguments supplied to the object define a set of addresses to which corresponding messages can be routed.

2.1. PDSynth architecture

The development of the toolkit, unfortunately, is still in an initial phase, however it already provides a minimal series of modules that can be easily used to create and process sounds. At the beginning of the project, in fact, after the first phases of software development we decided to move the attention to the design and the construction of hardware devices to be used in combination with the toolkit. The modules currently available can be classified into three distinct categories (Signal generators, Filters, Envelope generators) which will be presented in detail below.

2.1.1. Signal generators

PDSynth currently offers five sound generation modules that emulate the behavior of classic analog synth oscillators. The modules offer the possibility of generating the following waveforms:

GENPULSE — band-limited pulse train generator;

GENSAWTOOTH — band-limited saw tooth wave generator;

GENSIN — sine wave generator;

GENSQUARE — band-limited square wave generator;

GENTRIANGLE — band-limited triangle wave generator.

The `GENSIN` module uses the Pure Data native object `osc~` for generating the sine wave. All other modules are based on reading data saved in tables (Wave Table Synthesis) [10].

The image in Figure 1 shows the patch of the `GENPULSE` module. The sound is generated by using the Pd object `tabosc4~`. It allows to read the data saved in a table using a polynomial interpolation of the third order (four points interpolation). The objects that manage the OSC messages are placed in the top right corner of the patch. The first object (`r OSCMessages`) is used to receive OSC messages sent in "*broadcast*" within Pure Data's native message infrastructure. The following object (`routeOSC /$1`) selects and sends on its leftmost outlet only the messages that have as the first tag of the address the name of the module. This can be defined during the creation of a new instance of the `GENPULSE` through the first argument of the patch (for example *Pulse1* in the upper part of the patch in Figure 1). The next `routeOSC` object allows to route properly the data related to the various parameters to its different outlets accordingto the OSC name address (`/Freq` — frequency, `/Amp` — amplitude, ...).

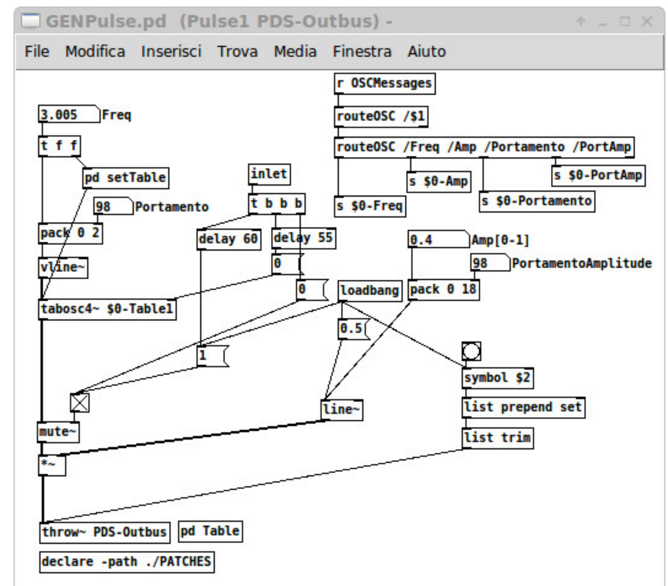

Figure 1: *GENPULSE - patch of the pulse train generator.*

The toolkit is based on digital sound generation techniques, so there is no real difference between audio band signal generators and LFO (Low Frequency Oscillator). Therefore, all generators can be used both in a frequency range below the threshold of audibility, as is typical for LFOs, and to produce audible sounds. For this reason, to achieve the generation of different waveforms we tried to make a compromise between the problems related to aliasing and the creation of signals with a frequency spectrum as wide as possible. After some initial experiments aimed at evaluating different approaches, we finally chose to generate band-limited signals by reading the waveform data stored in different tables. In order to be able to produce spectra that are very rich in harmonics, we decided to divide the audible frequency spectrum into eight distinct regions, each corresponding to a table containing the waveform with a suitably calculated harmonic frequency content to avoid aliasing phenomena.

The image in Figure 2 shows the subpatch of the `GENPULSE` module in which the eight tables are defined. Each table is related to a different region of the frequency spectrum: `$0-Table1` contains a waveform made of 511 partials that is used at the low end of the spectrum. While, at the opposite, `$0-Table8` contains a waveform generated using only three partials which is used to generate the sound in the upper part of the frequency spectrum.

All the signal generators, except the sinusoidal oscillator, use this approach based on the reading of eight tables with waveforms characterized by a different bandwidth. The change of the frequency parameter determines the selection of the appropriate table to be used for reading. The image in Figure 3 shows the `setTable` subpatch. It receives as input the frequency value in Hz and, by means of the conditional structure `if` contained in the object `expr`, it controls which is the table to be read according to the interval of frequencies in which the oscillator is called to operate.

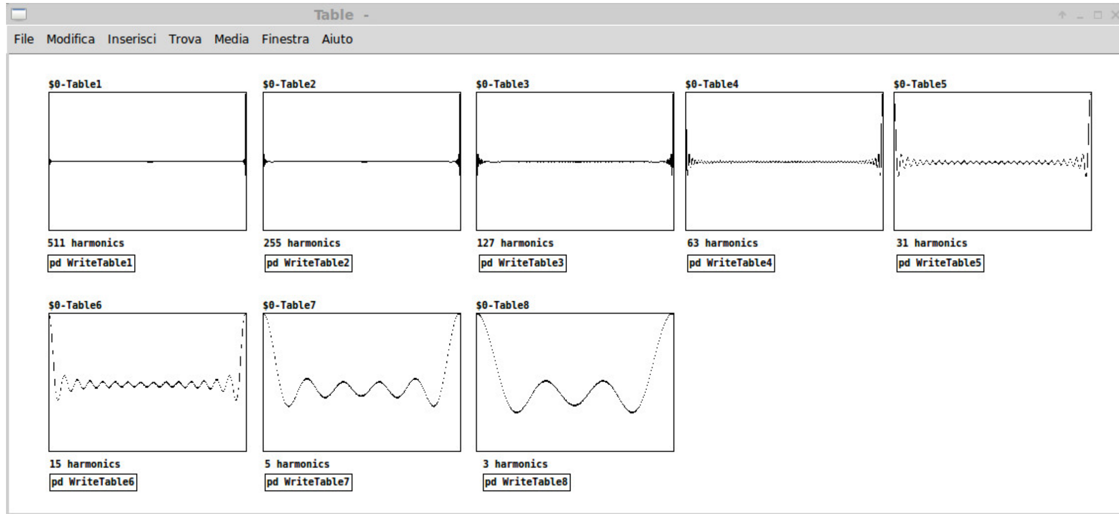

Figure 2: *GENPULSE — eight tables used for WaveTable synthesis.*

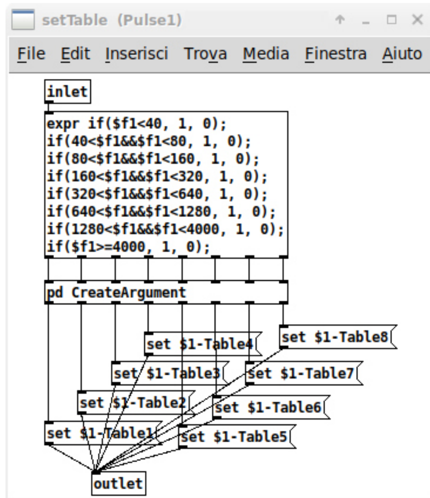

Figure 3: *GENPULSE — the setTable subpatch.*

The content of the wavetables shown in Figure 2 can be generated by using the Pd command `cosinesum` that allows to create a wave according to a sum of cosines harmonics (`sinesum` is the command in Pd to realize the sum of sine waves). The image in Figure 4 shows five subpatches used to create as many wavetables, each with its own number of partials (ie. `WriteTable8` — three partials, `WriteTable6` — fifteen partials, ...). The number placed after the `cosinesum` command define the length of the table to be generated expressed in number of points. In the image it is possible to notice that the number of points in the tables varies with the number of partials. In fact, as reported in the Pure Data manual, it is better to use tables composed of 512 points for waveforms containing up to fifteen partial. Above this threshold it is convenient to calculate the length of the table L as the number, power of two, greater than the product shown in equation 1, where N_p indicates the number of partial

$$L > 32 * N_p. \tag{1}$$

For example, to generate `$0-Table1` containing 511 partial we need to use 16384 points. To calculate the frequency values to be used for changing the table to be read as a function of the frequency value, it is possible to observe that from the previous relation the maximum number of partials can be obtained according to the size of the table. This value can be calculated by inverting the previous relationship and subtracting one as a safety margin as shown in equation 2:

$$N_p = \frac{L}{32} - 1. \tag{2}$$

Once this value is known it is possible to obtain the maximum reproducible frequency through the equation 3:

$$f_{max} = \frac{20000}{N_p} \tag{3}$$

it was decided to use the frequency of 20000Hz (with respect to the theoretical value of the Nyquist frequency equal to 22050Hz for the standard sampling frequency of 44100Hz) as an additional safety margin with respect to the occurrence of aliasing phenomena. Returning to the example of `$0-Table1`, we obtain therefore:

$$f_{max1} = 20000/511 = 39,1 \tag{4}$$

as it is visible in the image in figure 3 the first table is changed for a frequency equal to 40 Hz. This same procedure has also been applied for the calculation of all the other values defined to realize the change of the table to be read using the object `tabosc4~`.

The pulse waveform can be generated with a series of cosines in which all the partials have a uniform amplitude distribution. The value of the amplitude a_n can be computed according to equation 5, where N is the maximum number of partials [11].

$$a_n = \frac{1}{N}. \tag{5}$$

As shown in Figure 4, the command `cosinesum` is followed by the number of points of the table and by a list of numbers that defines the amplitude factor of each partial. In the case of the pulse waveform, this numbers are all the same and equal to the inverse of the number

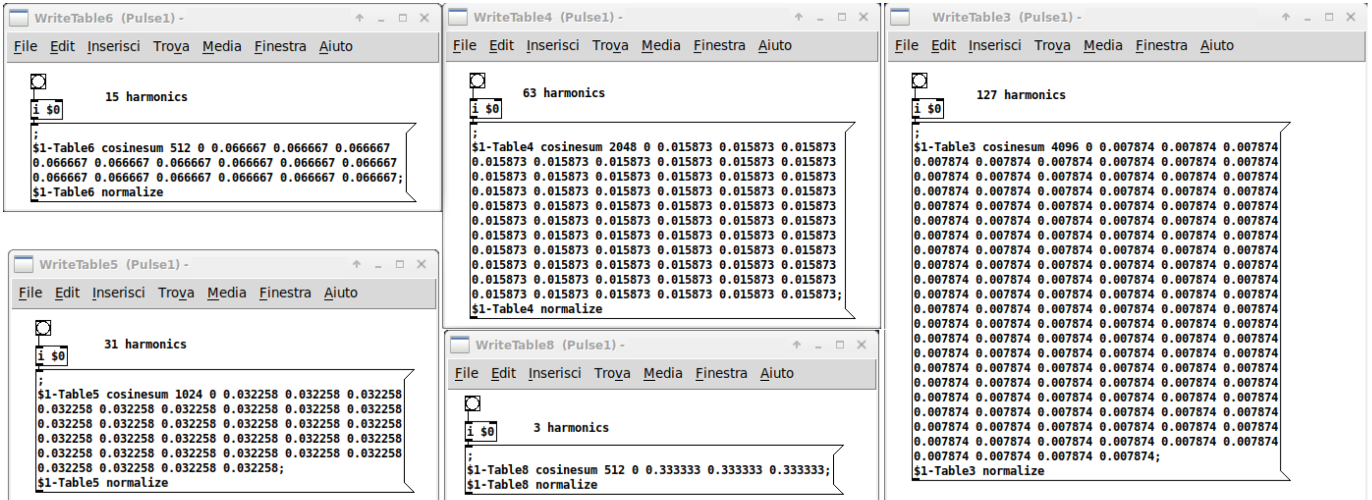

Figure 4: *GENPULSE — commands to create five wavetables.*

of partials. Creating this lists of amplitude coefficients for all the different waveforms is a long and repetitive process not so easy to do by hand. This is even more true, when it is needed to generate a very large number of partials as in the case of `$0-Table1`, as shown in the top right corner of Figure 2.

For this reason a Python script has been developed in order to automate the creation of a text file containing the list of amplitude coefficients. The script created for the pulse generator is shown below. In the first line, inside the `open` function, it is necessary to define the name of the text file where data will be stored. On the next line, the `MaxOrder` parameter defines the maximum number of partials to be generated. The content of the text files generated by this script can be easily copied and pasted into the Pure Data messages (see figure 4) used to populate the tables with the various waveforms.

```
out_file=open("Coef-Pulse.txt","w")
MaxOrder=511
x=round((1./MaxOrder),6)
out_file.write(str(0)+" ")
#The DC component is equal to 0
for i in range(1,MaxOrder+1):
        out_file.write(str(x)+" ")
out_file.close()
```

2.1.2. *Filters and sound processing*

In addition to sound generation modules, PDSynth provides also patches implementing filters. So far, three different filters of the fourth order have been created:

FLTBandPass – band pass filter based on the Pd object `vcf~`;

FLTHighPass – high pass filter based on the Pd object `hip~`;

FLTLowPass – low pass filter based on the Pd object `lop~`.

Figure 5 shows an example based on the `FLTBandPass` module. The patch realizes a small bank of filters, composed of three modules placed in parallel, used to filter white noise. Each filter is identified through a different OSC namespaces (`BP1`, `BP2`, `BP3`).

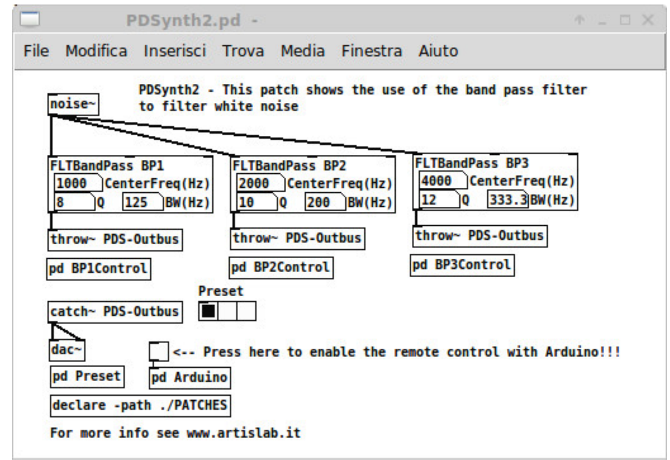

Figure 5: *Example patch of the FLTBandPass module.*

The structure of the filters patches is analogous to those of the generators with regard to the internal management of OSC messages. The filters differ from the generators only because of the presence of an audio inlet used to provide the input signal to be processed.

2.1.3. *Envelope generators*

Two modules were also developed to generate envelopes:

ENVTable — envelope generator defined through a table;

ENVADSR4 — ADSR type envelope generator with fourth order polynomial interpolation.

The `ENVTable` module allows you to generate a time envelope by reading data contained in a table. Python scripts have been created for the generation of envelope tables with different temporal trends. The image in Figure 6 shows different two traits (attack and release) envelopes generated by Python.

The `ENVADSR4` module, instead, realizes a four-state ADSR envelope with fourth order polynomial interpolation. As shown by

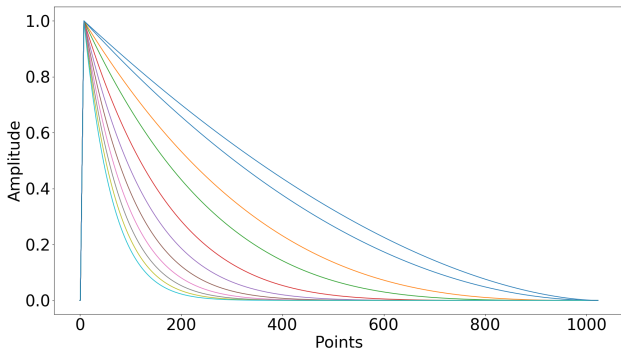

Figure 6: *Two piece polynomial envelope.*

Puckette [10] this type of interpolation allows to obtain a trend very similar to the logarithmic one, but with a reduced computational cost and a greater simplicity of implementation. As is known, the logarithm function diverges towards less infinite when the argument is approaching towards zero, which makes serious precautions necessary in the realization of logarithmic envelopes through truncation or approximation processes. On the contrary, the use of polynomial interpolation eliminates this problem by also offering the possibility of modifying the slope of the curves in a very simple manner by varying only the order of the polynomial used.

2.2. PDSynth-00

Starting from the initial idea to develop an exclusively software environment, we tried to use DIY controllers based on the Arduino prototyping platform to control the modules of the toolkit. This first experiments encuraged to broaden the vision of the toolkit by incorporating, therefore, both software development and the design of hardware devices to control the software architectures. The intent of the project has therefore been transformed into the creation of an environment for prototyping and developing portable electronic musical instruments and synthesizers.

On the hardware side, the project was oriented towards the realization of physical devices, equipped with potentiometers, sensors and other interaction systems. *PDSynth-00* (see Figures 7 and 8) is the first DIY prototype of a controller made by Artis Lab, in Spring 2016, that is born from the idea of an hardware device useful to control the synthesis and sound processing architectures created with the PDSynth toolkit.

Figure 7: *Rear panel of PDSynth-00.*

PDSynth-00 is a reprogrammable electronic musical instrument that can perform different functions depending on the software that is loaded into the Arduino board. It is equipped with six slide potentiometers and twelve buttons. Everything is contained in a simple and light container made of plywood shaped using a laser cutting machine.

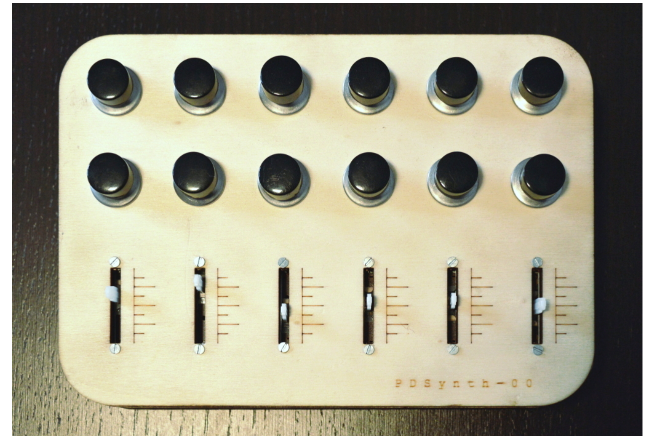

Figure 8: *Top view of PDSynth-00.*

PDSynth-00 can be interfaced with Pure Data through the Firmata protocol. By this way the data relating to the position of the six cursors and the status of the buttons can be sent to the program listening on the serial port and used to perform action or modify parameters inside the PDSynth patches.

3. SYNTHBERRY PI

SynthBerry Pi was born as a natural evolution of the PDSynth-00. The Arduino prototype is not autonomous, it is only useful for controlling the PDSynth modules running on a computer. SynthBerry Pi, enstead, integrates controller and computer through the use of a Raspberry Pi mini computer allowing to create an autonomous device able to generate sound that can be modified via a control surface.

3.1. The control surface

SynthBerry Pi is equipped with an hardware control interface consisting of eight slide potentiometers. The prototype was built, like the previous one, using two panels of plywood shaped with a laser cutting machine. Figure 9 shows the front view of the prototype.

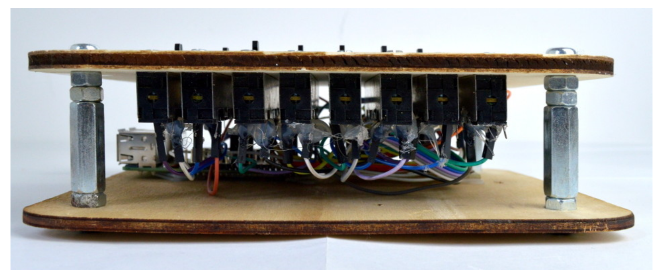

Figure 9: *The front view of SynthBerry Pi.*

The slide potentiometers are mounted on the front panel of the device, the assembly between the two panels of the prototype was

carried out through hexagonal steel spacers of suitable length. The image in Figure 10 shows the top view of the prototype.

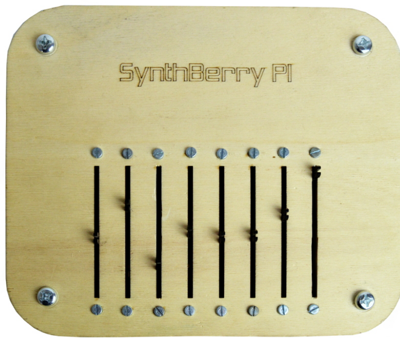

Figure 10: *Top view of the prototype.*

3.1.1. Hardware set up

Raspberry Pi is not equipped with analog to digital converters (ADC) allowing the connections of potentiometers. For this reason, the analog to digital converters *MCP3008* was used to read the voltages related to the positions of the slide potentiometers. The *MCP3008* is an integrated circuit that provides eight analog input channels with 10 bit digital resolution. Figure 11 shows the simulation of connections among the various components using a breadboard, while Figure 12 shows the circuit schematic. For simplicity, only one potentiometer has been inserted since all the others must be connected in a similar way to the ADC inputs.

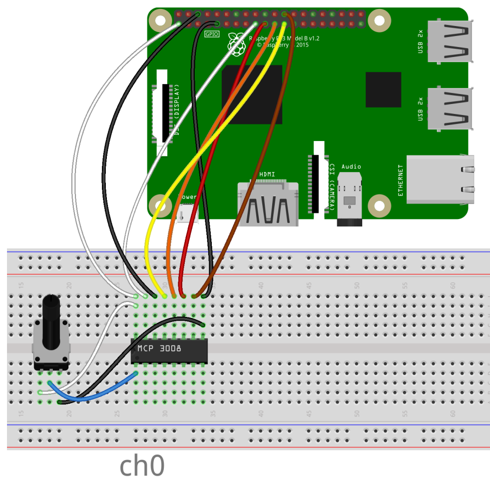

Figure 11: *Simulation of connections using a breadboard.*

The communication between Raspberry and the ADC is based on the *SPI (Serial Peripheral Interface)* serial communication protocol. SPI is a communication system between a microcontroller and other integrated circuits or between multiple microcontrollers. It is a communication standard, created by Motorola, in which the transmission takes place between a control device (called *master*) and one or more controlled devices (called *slave*). The master device controls the communication bus, emits the clock signal and decides when to start and end communication.

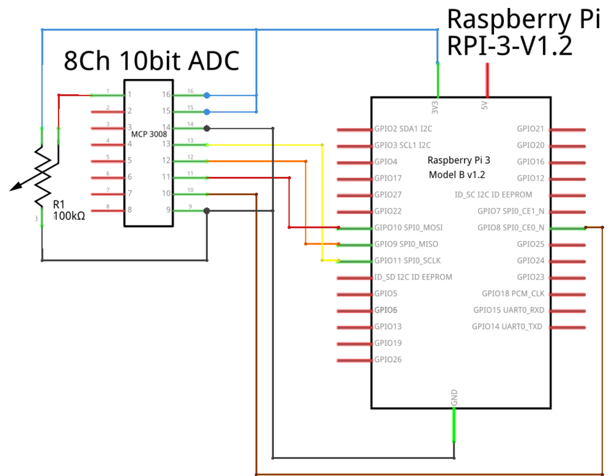

Figure 12: *Circuit schematic.*

The SPI communication system is commonly defined as four-wire, since for the transmission of data four distinct signals are generally used:

- **SCK**: Serial Clock (emitted from the master)
- **MISO**: Serial Data Input, Master Input Slave Output
- **MOSI**: Serial Data Output, Master Output Slave Input
- **CS**: Chip Select, Slave Select (issued by the master to choose which slave device to communicate with).

Chip Select is the only connection that is not always necessary in all applications since its just needed to manage multiple slave devices. A connection that defines the reference level of the voltage, often referred to as **GND**, must be added to these four wires.

3.1.2. Software set up

The reading of the data acquired by the ADC is realized through a Python script, which uses the *SPIDEV* library for the management of SPI devices. The following code shows a fragment of the Python script with the commands necessary to open the communication with the ADC and to perform a reading of the data through the `ReadChannel()` function. The `spi.xfer2()` function is invoked, inside `ReadChannel()`, to request the ADC to read the voltage value of a given channel.

```
#Open SPI bus
spi=spidev.SpiDev()
spi.open(0,0)
spi.max_speed_hz=1000000
#Function to read SPI data from MCP3008 chip
#Channel must be an integer 0-7
def ReadChannel(channel):
  adc=spi.xfer2([1,(8+channel)<<4,0])
  data=((adc[1]&3)<<8)+adc[2]
  return data
```

The transfer of the data read by the analog to digital converter, between the Python script and Pure Data, is achieved sending, on a specific port, local network messages. For this purpose we use the `pdsend` program provided within the standard Pd package, used as

a sub-process within the Python script. The following code shows the creation of the `subprocess p` which invokes the `pdsend` program used to send data on port 9000 of the local computer. The `send2Pd()` function is used to send messages through `pdsend`. The last line shows the use of the `send2Pd()` function that take as argument a string composed by the concatenation of two numeric values: the first to define the channel of potentiometer and the second to provide the ADC reading.

```
#Create a subprocess to send data to Pd
p=subprocess.Popen(["pdsend","9000"],
        stdin=subprocess.PIPE)
#Define the function to send data to Pd
def send2Pd (message=' '):
  print >> p.stdin, message
#How to use the function to send data to Pd
send2Pd('0'+str(pot_Volts0)+';')
```

A simple protocol was designed to send messages from Python to Pure Data keeping the data of the different potentiometers separate and easily differentiable. A list of two numbers is sent, the first is a label (from 0 to 7) useful for identifying the potentiometer, the second number is the numeric data obtained from the reading made by the digital converter. To receive data in Pure Data the `netreceive` object is used which opens a server listening on the port corresponding to that used by `pdsend`. The expedient used in the construction of the message sent by Python simplifies the sorting of data that can be easily accomplished through the native object of Pd `route`.

The data acquired by the ADC are filtered to reduce random fluctuations due to noise through the use of an average filter that generates an average output value every ten converter readings. The following code shows the simplified structure for reading and transmitting data of a single potentiometer. Within an infinite cycle, the `Count` counter is incremented and the values supplied by the ADC are read, divided by 1023 (to obtain numbers between 0 and 1) and accumulated on the `readPot0` variable. Every ten readings the `readPot0` variable is divided by ten, obtaining the average value. If the new value has undergone a variation compared to the previous one greater than 0.5%, the value of the `pot_Volts0` variable is updated. The value of this variable is then sent to Pd through the `send2Pd` function. After that the values of the counter and the accomulation variable are both reset to zero and the program execution is suspended for a time defined by the `delay` variable.[4]

```
while True:
 readPot0+=ReadChannel(pot0_channel)/float
     (1023)
 Count+=1
 if Count==10:
   readPot0=0.1*readPot0
   if abs(pot_Volts0-readPot0)>0.005:
        pot_Volts0=readPot0

   send2Pd('0'+str(pot_Volts0)+';')
   readPot0=0
   Count=0
 # Wait before repeating loop
 time.sleep(delay)
```

[4]In this way, using a value equal to $0.01s$ for the variable `delay`, the reading is made every $10ms$ and a new value is sent to Pure Data every $100ms$.

A script was created to start both Pure Data and the Python program to manage the ADC. Since we want to use the prototype as a common electronic instrument through the use of only the control surface we have chosen to use the Raspberry Pi *headless* without the connection of screen, mouse and keyboard. For this purpose, the script must be started automatically during the startup phase of the Raspberry Pi. To do this, a special service, launching the start script, has been created and set up to be started automatically during the initial phases of execution of the operating system.

3.2. The audio engine

The audio engine of the prototype is based on the use of a modified version of the first PDSynth sample patch. The patch offers the possibility to separately control the amplitude and the frequency of three oscillators generating different waveforms (square wave, pulses train and sawtooth wave). Furthermore, it provides a delay line with a feedback path. The delay line can be controlled by two parameters that can be modified in real time through the potentiometers of the prototype control surface: the delay time and the feedback coefficient. The eight potentiometers of the prototype have been associated to likewise control parameters of the patch according to the following scheme:

- A0 - square wave oscillator frequency;
- A1 - amplitude of the square wave oscillator;
- A2 - pulse generator frequency;
- A3 - amplitude of the pulse generator;
- A4 - frequency of the sawtooth oscillator;
- A5 - amplitude of the sawtooth oscillator;
- A6 - delay time;
- A7 - delay feedback.

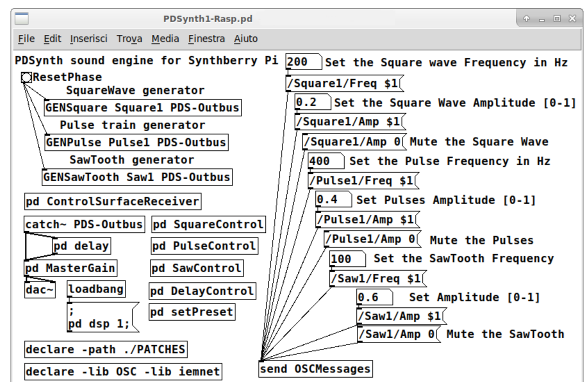

Figure 13: *Pd patch of the audio engine.*

Figure 13 shows the audio engine patch and the use of three signal generation modules. The modules are placed in the top left corner of the patch. The first argument of each object is a name that is used as an identifier for addressing OSC messages (*Square1*, *Pulse1*, *Saw1*). The second argument provided in the creation of the sound generators (*PDS-Outbus* in the image) allows to define the name of the *bus* on which the audio signals produced by the various generators will be accumulated. In this case, all the signals are collected by the `catch~ PDS-Outbus` object and sent both to the delay patch

and to the sound card output through the `dac~` object (in the lower left part of the patch in Figure 13).

On the right side of the patch, the *number boxes* allow to change the amplitude and the frequency of the various oscillators by sending OSC messages through the `send OSCMessages` object. The image shows also how to create the OSC message addresses to control the parameters of the modules.

3.3. Eurorack Module

In autumn 2019 a new version of the prototype was created in the form of a 18 hp eurorack module. The image in Figure 14 shows the front view of the module.

Figure 14: *The front view of the module.*

Figure 15 shows the internal structure of the module. An handcrafted PCB board is connected on the GPIO pins of the Raspberry Pi. The potentiometers are connected to the PCB where the ADC is also housed. This second prototype made in the form of a Eurorack module is completely analogous to the first prototype both in terms of hardware and software.

4. CONCLUSIONS

In this work we have presented *SynthBerry Pi* an autonomous synthesizer based on Raspberry Pi and Pure Data. The PDSynth toolkit was used as audio engine of the prototype. This toolkit provides a series of Pd patches that can be easily used as modules to create high level architecture to generate and process sounds. To run the PDSynth synthesis architectures a Raspberry Pi mini computer was used; a control surface made up of eight slide potentiometers was build to provide a suitable hardware device to play the instrument controlling in real time the sound parameters.

SynthBerry Pi was used in several live performances and also in studio recordings. In future work we intend to add a second ADC to the prototype to have eight more input channels useful for implement the control voltage (CV) of the synth parameters. Furthermore, we intend to create a more intuitive and powerfull control surface by also adding buttons, LEDs and rotary encoders.

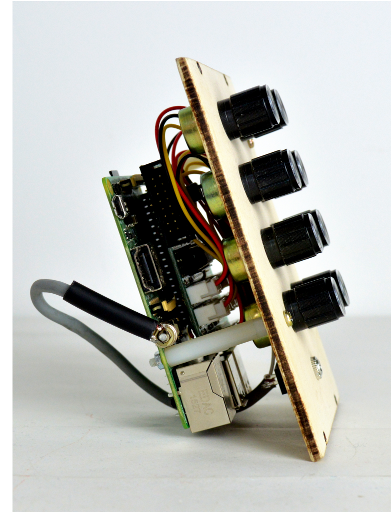

Figure 15: *The internal structure of the module.*

5. REFERENCES

[1] J. Noble, *Programming Interactivity. A Designer's Guide to Processing, Arduino and openFrameworks*, OReilly, Sebastopol, CA, 2009.

[2] R. Wilson, *Make: Analog Synthesizers*, Maker Media, Sebastopol, CA, 2013.

[3] J. Reuter, "Case study: Building an out of the box Raspberry Pi modular synthesizer," in *Proceedings of Linux Audio Conference (LAC14)*. Karlsruhe, 2014.

[4] F. Meier, M. Fink, and U. Zölzer, "The JamBerry - a standalone device for networked music performance based on the Raspberry Pi," in *Proceedings of Linux Audio Conference (LAC14)*. Karlsruhe, 2014.

[5] V. Lazzarini, Timoney J., and Byrne S., "Embedded sound synthesis," in *Proceedings of Linux Audio Conference (LAC15)*. Mainz, 2015.

[6] H. von Coler and D. Runge, "Teaching sound synthesis in C/C++ on the Raspberry Pi," in *Proceedings of Linux Audio Conference (LAC17)*. Saint-Etienne, 2017.

[7] M. Puckette, "Pure Data," in *Proceedings of International Computer Music Conference (ICMC97)*, pp. 224–227. Thessaloniki, 1997.

[8] M. Wright, A. Freed, and A. Momeni, "Open Sound Control: State of the art 2003," in *Proc. of the 2003 Conference on New Interfaces for Musical Expression (NIME-03)*, pp. 153–159. Montreal, 2003.

[9] M. Wright, "Open Sound Control: an enabling technology for musical networking," *Organised Sound*, vol. 10, no. 3, pp. 193–200, 2005.

[10] M. Puckette, *The Theory and Technique of Electronic Music*, World Scientific, 2007.

[11] C. Dodge and T. A. Jerse, *Computer Music*, Schirmer, New York, 1997.