Politecnico di Milano

Euclidean Algorithm for auto-generative patterns in a Supercollider application

Project of Computer Music course

prof. Antonacci Fabio

Students:
de Bari Mauro Giuseppe Matr.899371
Albertini Davide Matr.883347
January 24, 2019

Abstract

This paper will describe the implementation of a drum-machine Supercollider app, based on Euclidean Rhythms.
The section 1 introduces concepts about Euclidean Rhythms, giving some references to already existing electronic modules that adopt them.
The section 2 concerns the historical development of the Euclidean Algorithm, from its first ancient origin to nowday developments for the musical world.
The section 3 focuses about the Supercollider application code, describing the Model-View-Controller pattern inside the application. For the sake of simplicity, it is divided in three subsection.
In the section 4 issues about the installation and configuration of the app are given. Morover some possible improvements are detailed.

1 Introduction

Euclidean rhythms are a large class of rhythms which can be generated through the Euclidean Algorithm. Making its first appearence in Euclid's Element, it's one of the oldest algorithm, and it was used to compute the greatest common divisor of two given integers. It was Godfried Toussaint to discover, in 2005, that the Euclidean algorithm's structure may be used to generate the aforementioned class of rhythms. Moreover, he found that those patterns are often part of traditional and world music[1]. He noticed that the main characteristic of those rhythms is that their onset pattern are distribuited as evenly as possible. Since then, Euclidean rhythm generators started to spread in the music production world, since with very few parameters one can generate very complex patterns and polyrhythms. A lot of eurorack synthesizer modules have been developed: Mutable Instruments Yarns[2], vpme.de Euclidean Circles[3] and 2HP Euclid[4], just to name a few. This paper will present a software implementation of a drum machine capable of generating Euclidean rhythms, using Supercollider programming language.

2 The algorithm

The purpose of the researchers in the past year was to find an easy way to generate all the possible rhythms present in music, in order to extend the dictionary of Automatic Music Composition[5]. Before proceeding with the implementation of the drum machine, it's necessary to dwelve into the algorithm of Euclid and to explain what it has to do with rhythm generation.
In order to compute the greatest common divisor between two integers, the greek mathematician proposed this solution: the smaller number is repeatedly subtracted from the greater until the greater is zero or becomes smaller than the smaller, in which case it is called the remainder. This remainder is then repeatedly subtracted from the smaller number to obtain a new remainder. This process is continued until the remainder is zero[6]. The same procedure can be done more efficiently with divisions.

Algorithm 1 Euclid's Algorithm

```
1: procedure EUCLID(m, k)
2:     if k == 0 then
3:         return m
4:     else
5:         return EUCLID(k, m mod k)
```

The derivation of G. Touissant[1] comes from an analyisis over Bjorklund's studies on SNS accelerators[7] in nuclear physics. In this case, time is divided into intervals and during some of these intervals an onset is to be enabled by a timing system that generates pulses that accomplish this task. The problem for a given number n of time intervals, and another given number $k < n$ of pulses, is to distribute the pulses as evenly as possible among these intervals.
In our case the algorithm to develop take as input a tuple (k, n), where n represents the length of the played sequence and k the number of onsets inside the sequence. The output is a sequence of length n, where the k onsets are equally spaced inside the sequence. For example, if we consider the tuple $(4, 16)$, the result has to be a sequence like:

[1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0]

As Touissant explains in his work[1], with this mechanism we can produce all the possible rhythmic sequences, linking them to a specific musical genre(Figure 1). For instance:

- $E(2, 3) = [1, 0, 1]$ is a common Afro-Cuban drum pattern. For example, it is the conga rhythm of the (6/8)-time Swing Tumbao[8].
- $E(2, 5) = [1, 0, 1, 0, 0]$, when it is started on the second onset ($[1, 0, 0, 1, 0]$) is the metric pattern of Dave Brubeck's *Take Five* as well as *Mars* from *The Planets* by Gustav Holst[9].
- $E(5, 16) = [1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0]$ is the Bossa-Nova rhythm necklace of Brazil, started on the second onset[10].

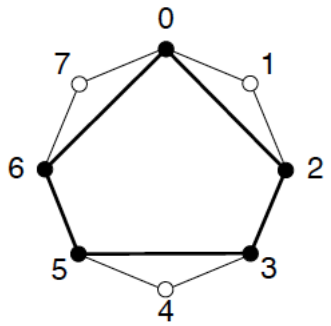

Figure 1: The Euclidean rhythm $E(5, 8)$ of the Cuban cinquillo

The application uses as reference the algorithm explained by Bjorklund[7]. This algorithm adopts the Euclidean's one(Algorithm 1), to calculate at each step a remainder,

needed to equally divide onsets inside the sequence of 0's and 1's.
Each cycle is composed by two steps: the first consists in creating the base string and the second in spacing the onsets inside the base string.
Considering m the length of sequence and n the number of onsets, and supposing that $n < \frac{m}{2}$, we need to create a sequence of $m - n$ zeros and one another of n ones. In the case of five pulses of thirteen cycles, we start from seqeuences [00000000] and [11111].
We start by simply dividing the five 1's into the eight 0's. This will put a 1 after every 0, with three 0's left over. It is visible that the three 0's correspond to the rest of division between 8 and 5: [0101010101] and [000].
Once again, we can think of this as two optimally distributed strings. One string is ten bits long and contains five "01" pairs (an optimal distribution of five pulses over ten slots). The second string contains three 0's. If we now distribute the three 0's over the five "01" pairs, we get three "010" triples with two "01" pairs left over: [010][010][010][01][01].
We still have two optimally distributed strings. "010010010" is an optimal distribution of three pulses over nine slots and "0101" is an optimal distribution of two pulses over four slots. We now repeat the process, dividing the two "01" strings into the three "010" strings. This gives us two five-bit strings ("01001") with one three-bit string ("010") as a remainder: [01001][01001][010][01].
We can stop the process when the remainder reaches one or zero. Our final pattern, therefore, is [0100101001010] , which is as evenly as we can distribute five pulses over thirteen slots.

3 The application

In the next tre chapter will be explained the structure of the application, with reference to Object Oriented Programming. The first section exposes the passage from the creation of sequence to reproduction of itself, the second one defines its humanization and the third concludes with an overview on the Graphical User Interface.

3.1 Sequencing

The first aim of the application is to play the sequences of onsets and downsets meanwhile the user is changing the values of the Supercollider application.
This is possible reproducing these three steps:

- Read new values related to length and onset of sequence
- Calculate new sequence
- Reproduce the calculated sequence for the sound selected by user

In order to following the principles of object oriented programming, it has been developed a class that contains the Euclidean Algorithm, necessary to the calculation of sequence. It is an adapted version of the algorithm discussed in the previous section and in [7]. The class contain the methods related to the building of string based on the remainder of division between length and steps. The first method calculates the array of remainders and counts related to each *level*. Considering that the number of downset (or 0's) is greater than the number of onsets (or 1's) it can be seen that the level -1 implies that

a 0 should be inserted in the string. Level -2 implies that a 1 should be inserted in the string. The "count" array tells us how many level $l-1$ strings make up a level l string. The "remainder" array is used to tell us if the level l string contains a level $l-2$ string. We use the remainder array both to keep track of the remainder of the previous division (it will become the denominator of the next division), and to determine how many levels deep we need to go. The process stops either when the remainder is zero (we have achieved a completely even distribution) or when the remainder is one (we have reached the end of the remainder series) [Code 1].
The same process can be applied also for sequences for which the number of 0's is less than number of 1's, only by considering the problem of equally spaced downsets inside onsets.

Code 1: Function to build final string of the Euclidean Algorithm

```
62 *build_string { | sequence, level, count, remainder, flag |
64         if(level == -1 , {
65                 sequence = sequence ++ [flag] //add 0 if k <= n/2;
66         }, {
67                 if(level == -2 , {
68                         sequence = sequence ++ [1-flag];
69                 }, {
70                         for(0, count[level]-1, {
71                                 sequence = this.build_string(
                                    sequence, level-1, count,
                                    remainder, flag);
72                         });
73                         if(remainder[level] != 0, {
74                                 sequence = this.build_string(
                                    sequence, level-2, count,
                                    remainder, flag);
75                         });
76                 });
77         });
79         ^sequence;
80 }
```

After the computation of the sequence, the application needs a way to continuosly reproduce the sequence, without stopping it if some parameters are changing, but instead uploading itself in real time. In order to this process to be perfeclty computed Supercollider provides an Event Stream 'manager' called *Pdef*. The *Pdef* incapsulates a certain type of *Pbind*, giving it a reference to a global value, defined as 'key', and reproduces it following the global clock definition. By changing some parameters of the *Pdef* or the entire *Pbind*(Code 3), this last uploads itself and proceeds with playing. The *Pbind* links the *SynthDef* that the user wants to play to 'tageted' strings that define the pattern. For instance it is possible to use the *Pbind* to play a sequence of notes of a *SynthDef* by defining an array of degrees, or midi notes, defining their duration or their amplitude and other features by simply using 'targeted' strings. From Supercollider library:

Code 2: Example of reproduction of a Pbind

```
// a SynthDef
SynthDef(\test, { | out, freq = 440, amp = 0.1, nharms = 10, pan =
    0, gate = 1 |
     var audio = Blip.ar(freq, nharms, amp);
     var env = Linen.kr(gate, doneAction: Done.freeSelf);
     OffsetOut.ar(out, Pan2.ar(audio, pan, env) );
}).add;
)

Pbind(\instrument, \test, \freq, Prand([1, 1.2, 2, 2.5, 3, 4], inf)
    * 200, \dur, 0.1).play;
```

In our case the *Pbind* contains the sequence that is playing and the *Pdef* globalizes it.

Code 3: Example of update for a Pdef

```
super.soundSource = Pdef(super.pdefId,
        Pbind(
                \instrument, \kick,
                \noteOrRest, Pif((Pseq(Array.fill(1024, {0}), inf))>
                    0, 1, Rest)
        )
);

//...

updateSequence{ | sequence |
var seq = Pseq(sequence, inf);
Pdef(super.pdefId,
        Pbind(
                \instrument, \kick,
                \noteOrRest, Pif(seq > 0, 1, Rest())
        )
);
```

3.2 Humanizer

Sometimes computer generated rhythms can sound too artificious, due to the fact that the software usually places the onsets of the various sounds perfectly on the grid given by the metre. A human player, on the other hand, has not a perfect timing, but plays with small delays instead of sitting perfectly on the time grid. In order to give the rhythm a more human feel, that we like to hear, there is the need of simulating those small timing imperfections. The key idea of this method is to have a high clock resolution, that allows to place the onsets in subdivisions of rhythms that are too short for being taken into account even by a professional drummer, but that still give the impression of a human playing.
In the actual implementation, the clock has been set to a resolution of 1024 cycles for each 4/4 bar. Since the Euclidean rhythm generator is generating patterns in terms of

16th notes, this means that each sequence step is actually 64 clock cycles long. It has been decided to keep the Euclidean pattern generation independent to the actual clock resolution, in order for the source code to be more reusable. This means that every time an Euclidean rhythm is generated, considering that it is expressed as a sequence of onsets or rests in 16th notes, it must be converted to a higher clock resolution, in order to open the possibility for generating small delays and approximate a humanized playing style [Code 4].

Code 4: Humanized Sequence

```
sequence16 = EuclideanRhythmGen.compute_rhythm(seqHitsKnob.value,
   seqLengthKnob.value); //in 16th notes
sequenceStraight = EuclideanRhythmGen.convertToClockResolution(
   sequence16, 256);
sequence = RhythmEditor.humanize(sequenceStraight, 256,
   humanizerKnob.value);
}
```

The first line computes the Euclidean sequence as an array of 0 an 1, representing onsets and rests in 16th notes. The sequence is then converted to the augmented clock resolution, the second parameter to be passed is the resolution of a quarter note in clock cycles. This parameter is passed from the outside so that the actual implementation of the conversion can be reused with other clock resolutions. The third line calls the method which randomly shifts by a small amount the onsets, generating a human feel. It is important to notice that the sequence, once converted to the actual clock resolution, is saved in a variable called sequenceStraight. This passage is crucial, since the humanize method is scheduled to be referenced every 4/4 bar, in order to be realistic and not to become a periodic humanization, which makes no sense at all. The next call to the humanize method, if the Euclidean rhythm parameters are not changed, must work on the previously saved sequenceStraight sequence, and not on the humanized sequence computed at the previous 4/4 bar, otherwise the original rhythm would diverge to something else [Code 5].

Code 5: Humanizer Function

```
1  RhythmEditor{
2
3
4          *humanize { | sequence, clockRes, amt |
5
6                  var sequenceConverted = Array.fill(sequence.size(),
                      {0});
7              var step_displacement, displaced_index, step16_index,
                   nSteps, timeNow;
8
9                  nSteps = sequence.size()/(clockRes/4); //number of
                      steps in 16th (equal to numHits parameter of the
                      euclidean sequence)
10
11                 for(0, (nSteps-1), { arg seqStep;
```

```
13                          step16_index = seqStep*(clockRes/4);
                                                //position of each 16th
                               step, according to the augmented clock
                               resolution

15                          step_displacement = round(rrand(-16,16)*amt)
                               ;        //displacement to be applied to
                               humanize the straight sequence
16                          displaced_index = step16_index +
                               step_displacement;  //new index at which
                               we write the displaced (humanized) onset

18                          if( displaced_index >= 0,
19                                  {
20                                          sequenceConverted =
                                               sequenceConverted.put(
                                               displaced_index, sequence
                                               [step16_index]);

22                                  },
23                                  {
24                                          sequenceConverted =
                                               sequenceConverted.put(
                                               step16_index -
                                               step_displacement,
                                               sequence[step16_index]);
25                          });

27                  });

29                  ^sequenceConverted;

31          }

33  }
34  }
```

3.3 Graphical User Interface

The Gui(Figure 2) is divided in two main sections:

- The instrument grid
- The side bar

The Instrument Grid is composed by an arbitrary number of slots, which can contain an instance of an instrument. The type of instrument can be selected through a drop down menu.

Each instrument, shows some specific parameters that act on the temporal and timbral characteristics of the sound, and a sequencer section that is common among all the instruments.

The sequencer section contains all the parameters useful for setting an euclidean sequence to the selected instrument, a humanizer knob, and a volume knob. As far as the Euclidean sequence is concerned, the step knobs, sets the length of the rhythmic sequence in terms of sixteenth notes; the hits knob sets the number of onsets, to distribute as evenly as possible in the sequence, according to the Euclidean algorithm. Finally, the offset knob allows the sequence to rotate, to achieve all the possible rotation of the same Euclidean rhythm. The humanizer knob can be used to specify the amount by which every onset in the sequence is shifted apart with respect to their precise location, in order to simulate a human behaviour. The volume knob simply allows to adjust the amplitude of each sound generated.

The Side Bar is used to set the global tempo in beat per minutes.

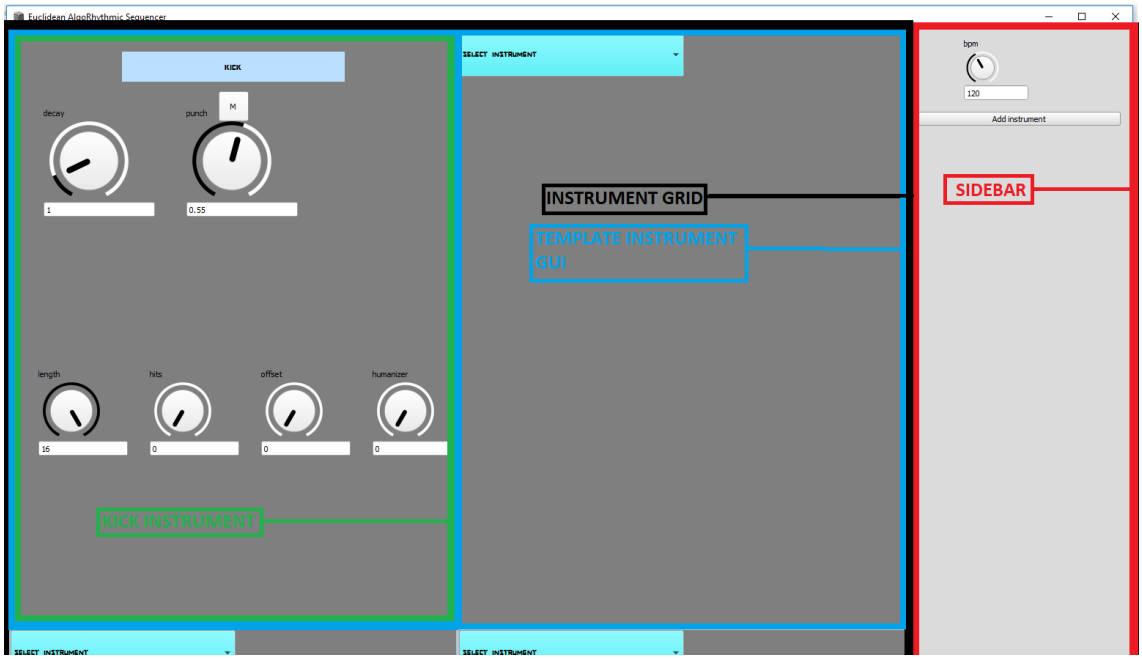

Figure 2: View of application

The graphical user interface has been developed following two guidelines: to be responsive and dynamic.

The responsiveness of a GUI means its adaptability to different screen sizes and formats. In order to achieve this goal, it has been decided to make a strong use of the composite pattern implemented in Supercollider. The composite pattern is characterized by a given class A, that can contain a list of other instances of the same type A, thus creating a three-like structure. As an example, the root could be the application window, that contains other groups of elements, that can contain themselves other elements and so on... Supercollider implements the composite pattern through the View class, in particular

each GUI object must belong to a subclass of View.

View requires two arguments, the container of the element, and a set of parameters describing its bounds in term of pixels.
In the actual implementation each GUI component passes itself to the constructor of the children components, so that they can infer its dimensions and scale their size accordingly.
In the code snippet below [Code 6], [Code 7], it is shown an example of this recurring pattern: a container, called instGrid, passes itself to its child component TemplateInstGUI. This behaviour is needed, because all the classes in the application that create GUI components, being subclasses of View, need the container View on which they must be attached to. It is also useful for the scaling purposes explained before.

Code 6: Generate View from main.scd

```
1   (
2   s.waitForBoot{(
    //Code from main.scd

159 instGrid = ScrollView.new(w,Rect(0,0, instGrid_width, w_height) );
    //...

173 slot1 = TemplateInstGUI.new.create(instGrid);
```

Code 7: Controller of TemplateInstGui

```
1   TemplateInstGUI{
2           var m, templateView;

4           create{ arg instGrid;

6                   var instGrid_width = instGrid.bounds.width;
7                   var templateView_spacing = 7;//spacing between the
                        various templateViews

9                   //CREATES THE CONTAINER "templateView"
10                  templateView = CompositeView.new(instGrid, Rect(0,
                        0, (instGrid_width/2) - templateView_spacing, (
                        instGrid_width*2/3) - templateView_spacing ) );
                    //...

30          }
31  }
```

To ensure complete scalability of the GUI, in the main.scd script, the sizes of the screen are retrieved through a system call, and those quantities are used to scale every other portion of the GUI. Going up in the hierarchy of components and subcomponents we reach the main window that contains the whole application, which is proportional to the screen size. This implementation allows to have a user interface that scales with the

screen dimensions.

In order to have a dynamic GUI, that allows the user to create, erase and replace instances of the various instruments, it was necessary to define a slot, which could host any different kind of instrument. Consequently, a way to abstract on the way each specific instrument is created, was needed.
The proposed solution makes use of the factory method patter, which is a common design pattern in object oriented programming. This enables writing of subclasses to change the way an object is created (to redefine which class to instantiate)[11]. To implement it, it is necessary to make an abstraction on the way instrument objects that are created (Inst class) and provide a class that is responsible for the creation of such objects (Inst Factory). The class diagram in Figure 3 depicts the simplified architecture for instruments creation and behaviour. The Supercollider classes are omitted for the sake of clarity.

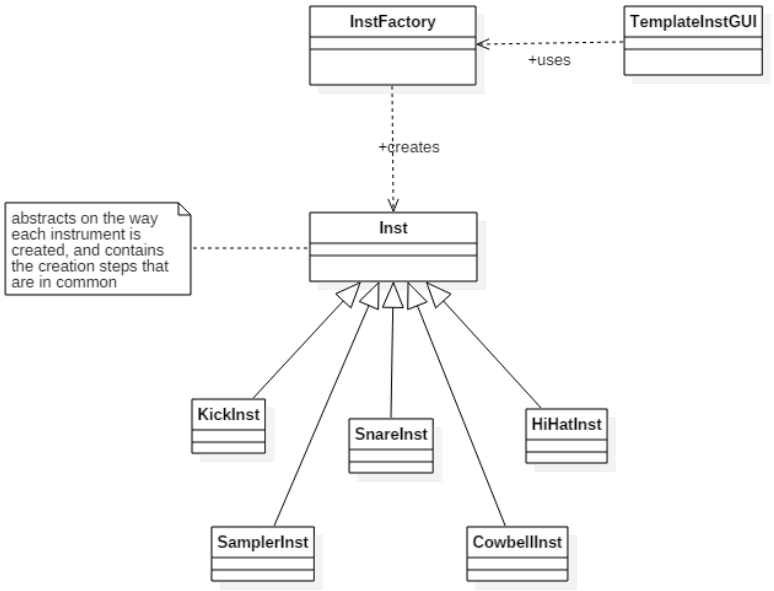

Figure 3: Class diagram of the application

To show the advantages of such pattern, it will be shown how TemplateInstGUI is able to create and "host" a generic instrument, by using InstFactory, without worrying about the differences in the creation of each specific instrument. Since Supercollider does not have interfaces, like most object oriented programming languages, the abstraction is done by dividing the common creational step among the various instruments in the method initializeSequencerGui in the superclass Inst. This method is then called during the creation of instrument specific GUI elements in the createView method, in each instrument class. In Code 8 only the code for the kick instrument is shown, since the other instruments present a very similar code structure.

Code 8: TemplateInstGui Class

```
1 TemplateInstGUI{
        //...
```

```
5           create{ | instGrid |
                //...

15              m = PopUpMenu(templateView, Rect(0, 0,
                   instGrid_width/4, instGrid_width/20));
16              m.items = InstFactory.getInstuments;
                //...

22              m.action = { arg menu;
23                      if(menu.item!="SELECT INSTRUMENT",
24                              {var inst = InstFactory.getIstance(
                                   menu.item);
25                                   inst.createView(templateView
                                      );
26                      }, {});
27              };
28          }
29  }
```

Code 9: Factory of Instruments

```
33  InstFactory {

35          classvar <>instrumentId=500;

37          *getIstance{ | name |

39                  var instance = case

41                  { name == "kick" }    { instance = KickInst.new}

43                  { name == "kick808" } { instance = Kick808Inst.new }

45                  { name == "snare" } { instance = SnareInst.new }

47                  { name == "SOSsnare" } { instance = SOSsnareInst.new
                        }

49                  { name == "hi-hat" } { instance = HiHatInst.new }

51                  { name == "sampler" }    { instance = SamplerInst.new
                        }

53                  { name == "cowbell" }    { instance = CowbellInst.new
                        }

55                  { name == "kalimba" }    { instance = KalimbaInst.new
                        }
```

```
57                  { name == "marimba" }    { instance = MarimbaInst.new
                        }

59                  { name == "SOShats"} { instance = SOShatsInst.new};

61                  instance.pdefId = this.prGenIstrumetId;
62                  ^instance;

64          }
            //...

67  }
```

Code 10: Instrument Abstract Class

```
69  Inst {
            //...

77          /*initializeSequencerGui INITIATES THE GUI RELATIVE TO A
               GENERIC INSTRUMENT*/
78          initializeSequencerGui { | instView |
                    //SETTING COMMON GUI DIMENSIONS
                    //SETTING INSTRUMENT LABEL
                    //CREATING EUCLIDEAN RHYTHM CONTROLS
                    //CREATING HUMANIZER CONTROLS

166         }
            //...

194 }
```

Code 11: Kick Class

```
197 KickInst : Inst {
            //...

202         /*CREATES THE INTRUMENT SPECIFIC GUI COMPONENTS, AFTER
               CREEATING THE COMPOSITE VIEW FOR THIS INSTRUMENT*/
203         createView{ | templateView |

205                 instView = CompositeView.new(templateView, Rect(0,
                       0, templateView.bounds.width, templateView.bounds
                       .height) );
206                 instView.background = Color.grey;

208                 super.initializeSequencerGui(instView);
                    //...

240         }
            //...
```

```
256 }
```

4 Conclusions

4.1 Further Improvements

There is always the opportunity to improve and expand the features of a software program, or at least this is true for our software.
An idea would be to insert in the rhythm creation chain, showed in the code snippet (numero), more sequence processing algorithm, such as a swing generator or a burst generator, which creates a burst of onsets with controllable density and duration. Those modifications became possible with the introduction of a high clock frequency that went beyond the basic subdivisions of 16th notes.
To include the functionality, some modifications on the humanize function are needed. In the actual implementation, the algorithm searches at each index position corresponding to a perfectly timed 16th step and, if there is an onset, this is shifted by a small amount, randomly. If new sequence processing units are added after the humanizer, the new algorithm would not know where to look for an onset. In general, each sequence processing element would need to know exactly where to look for onsets, based on the previous processing step, thus the sequence processing functions could not be interchanged with each other. To solve this problem, and making the implementation of each processing unit more straightforward, the sequence should be not just an array, but an object that saves not only the sequence itself, but the indexes of each onset in a separate array.
Another useful addition would be the export function. It could export the whole audio output, or the individual track for each instruments, or even MIDI tracks, to be used outside Supercollider.

4.2 Installation

To honour the object oriented paradigm, a class implementation has been preferred.
In order for the program to work correctly, it is not sufficient to run the code block in the main.scd file, since it is necessary to install all the classes that we developed for the application.
To add those classes in SuperCollider, it is necessary to follow these steps:

1. Locate the folder in your file system where SuperCollider looks when compiling user created classes, by using the command 'Platform.userExtensionDir';
2. Paste all the .sc file (class files) in that folder
3. Recompile the class library though the IDE by accessing from the tool bar 'Language -> Recompile Class Library'

References

[1] Godfried Toussaint, "The euclidean algorithm generates traditional musical rhythms," Tech. Rep., School of Computer Science, McGill University, Montreal, Quebec, Canada, 2004.

[2] O. Gillet, "Mutable instruments. yarns, eurorack for generetion of euclidean rythms," Available at https://mutable-instruments.net/modules/yarns/manual/.

[3] Vladimir Pantelic, "Euclidean circles eurorack," Available at https://vpme.de/euclidean-circles/.

[4] "Modulus for generetion of euclidean rythms," Available at http://www.twohp.com/modules/euclid.

[5] J.-P. Allouche and J. O. Shallit, Eds., *Automatic Sequences*, Cambridge University Press, 2002.

[6] Euclid, *Elements*.

[7] E. Bjorklund, "The theory of rep-rate pattern generation in the sns timing system," Tech. Rep., Los Alamos National Laboratory, Los Alamos, U.S.A, 2003.

[8] Tom Klower, *The Joy of Drumming: Drums and Percussion Instruments from Around the World*, Binkey Kok Publications, Diever, Holland, 1997.

[9] Michael Keith, *From Polychords to Pólya: Adventures in Musical Combinatorics*, Vinculum Press, Princeton, 1991.

[10] Gerard Behague, "Bossa and bossas: recent changes in brazilian urban popular music," *Ethnomusicology*, vol. 17, no. 2, pp. 209–233, 1973.

[11] Ralph Johnson John Vlissides Grady Booch Erich Gamma, Richard Helm, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Press, Boston, Massachusetts, U.S.A., 1994.