ALGOJUGGLEDRUMS - CREATING RHYTHMS FROM JUGGLING PATTERNS

Albert Gräf

Computer Music Dept.
Johannes Gutenberg University (JGU) Mainz, Germany
aggraef@gmail.com

Sophia Renz

Art History
Johannes Gutenberg University (JGU) Mainz, Germany
sophia.renz03@gmail.com

ABSTRACT

The paper introduces "algojuggledrums", a new algorithm for translating juggling patterns into musical loops which can be played back at a given tempo. We briefly explain the underlying notions related to juggling, discuss the main routine of the algorithm, and present an implementation which encompasses a Pd patch and an external written in Lua.

1. INTRODUCTION

Apart from some early forerunners in the 1930s and the 1950s, electronic drum machines came into their own in the 1960s with equipment by Wurlitzer, Seeburg, and Korg. Digital grooveboxes, their modern counterpart, have taken their place in music production and beat making since the ground-breaking work of Roger Linn in the 1980s which led to the creation of the LM-1 (1980), the LinnDrum (1982), and the AKAI MPC 60 (1988). Modern versions of the MPC and similar devices by other manufacturers are still prevalent in hip hop music, but they are also used in many other genres. While early drum machines let the musician choose from a number of presets for different common rhythms, modern grooveboxes let you program beats by employing a combination of MIDI sequencing and sampling techniques.

An alternative, lesser-known approach has been the algorithmic generation of rhythms which can be found, e.g., in Clarence Barlow's pioneering work on rhythmic "indispensabilities" [2], as well as in Godfried T. Toussaint's "Euclidean rhythms" which have become quite popular due to their implementations in both hard- and software [14]. Also worth mentioning here is the art of "change ringing" which can be described in terms of permutation groups [8]. In contrast to grooveboxes, which are programmed like MIDI sequencers, these algorithmic approaches are based on mathematical models which allow rhythmic sequences to be generated in a more or less automatic fashion.

This is also the approach taken in this paper. The novel aspect of the method presented here is that we employ so-called "siteswap" patterns, a numerical notation system that provides a simplified, theoretical representation of specific juggling tricks. While juggling is more of a sport or a visual art form, like music it also involves abstract and repeating patterns which are often performed in alignment with an underlying *beat*. By mapping siteswap patterns onto musical and rhythmic structures, we explore an alternative to motion-based performance tracking. The use of juggling movements as a control interface for electronic music has already been investigated in several prior projects [3, 13, 15, 16]. In contrast, our approach enables the generation and control of a wide range of sequences by basically adjusting siteswap parameters – without necessarily requiring the patterns to be physically jugglable.

The paper summarizes and builds on the Master Thesis (in German) by one of the authors (Sophia Renz). If you'd like to dive deeper, the thesis is available online for your perusal [12].

2. SITESWAP PATTERNS

Our algorithm employs a simplified model of periodic juggling patterns known as "siteswap" patterns [11]. These are characterized by the following properties:

- The objects (let's call them balls) are thrown at a constant beat, i.e., at equidistant points in time.
- At every beat, at most one ball gets caught, and that same ball immediately gets thrown again.
- When thrown, each ball stays in the air for a prescribed number of beats.
- Patterns are periodic, i.e., they repeat indefinitely.

Thus, the balls are caught and thrown sequentially at discrete points in time. In practice, most jugglers use both hands, thus alternating between the left and the right hand for each throw.

Under these conditions the juggling pattern can be denoted as a simple list of nonnegative numbers $a_1, a_2, ..., a_n$ which repeats indefinitely. Here each "throw" $a_i > 0$ denotes the number of beats the ball remains in the air until it gets caught and thrown again; a zero value $a_i = 0$ means that no ball gets caught or thrown at this point in the sequence. For instance, the pattern 5,0,1 indicates an infinite juggling sequence 5,0,1,5,0,1,..., where we first throw ball #1 at the first beat so that it stays in the air for 5 beats; in the second beat no balls get thrown; in the third beat ball #2 gets thrown for just 1 beat; in the fourth beat ball #2 lands and gets thrown for another 5 beats; in the fifth beat no balls get thrown; in the sixth beat ball #1 lands again and gets thrown for 1 beat; and finally, in the seventh beat, ball #1 gets thrown for 5 beats again, and the entire sequence repeats from this point on. The resulting sequence can be visualized in a so-called juggling diagram, cf. Fig. 1.

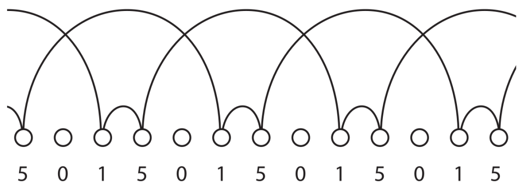

Figure 1: Juggling diagram [10, p. 3].

Not every list of nonnegative numbers is a valid siteswap pattern, however. For a quick check, we can calculate the average throw $m = (a_1 + ... + a_n)/n$. It turns out that this must be an integer which in fact gives the number of balls required to realize the juggling sequence. For instance, our previous example 5,0,1 has an average throw of (5+1)/3 = 2, which is an integer. In fact, as we've already seen, 2 balls are needed to realize this sequence. In contrast, the sequence 5,4,2 is not a valid siteswap pattern, because 5+4+2 = 11 and 11/3 is not an integer. But even if a sequence has an integer average throw value, it may still not be valid, because it may cause *collisions*, i.e., different balls landing at the same time. E.g., consider the sequence 5,4,3 which has an average throw of 12/3 = 4 and thus satisfies the average throw criterion. If we throw a ball at beat 1 then it will land again at beat 1+5 = 6, because the sequence starts with a throw of 5. But the second ball to be thrown at beat 2 has a throw of 4 and will thus land at the same beat 2+4 = 6, causing both balls to land at the same time, which is not permitted in a siteswap pattern.

There is an algebraic criterion which lets us decide easily whether any given sequence denotes a siteswap pattern. It involves constructing a test sequence $b_i = (a_i + i - 1) \mod n$ $(i = 1, ..., n)$. The sequence is a siteswap pattern if and only if the numbers b_i are all different [10]. Our sequence 5,4,3 yields the test sequence 5+0,4+1,3+2 which gives 2,2,2 (mod 3), thus 5,4,3 is not a siteswap pattern. On the other hand, the similar sequence 5,3,4 yields the test sequence 2,1,0 and thus is a siteswap pattern.

Siteswap patterns have many interesting mathematical properties, and there are algorithms to actually construct such patterns. These are beyond the scope of this paper, so we refer the reader to Polster's survey and his book instead [10, 11].

But let us briefly discuss the property that this kind of juggling patterns derives its name from. Consider a valid siteswap pattern $a_1, a_2, ..., a_n$ and two beat indices $1 \le j < k \le n$ such that $a_j \ge d = k - j$. Then it is always possible to construct a new valid siteswap pattern $b_1, b_2, ..., b_n$ such that $b_i = a_i$ for all $i \ne j, k$, $b_j = a_k + d$, and $b_k = a_j - d$ which effectively swaps the landing sites of the balls thrown at beats j and k. This is called a *site swap*.

Note that a site swap is always possible in the case of two adjacent positions $k = j + 1$ if position j is not empty, i.e., $a_j > 0$. In this case, $b_j = a_{j+1} + 1$ and $b_{j+1} = a_j - 1$. It is this special case which is discussed in [10, p. 6]. For instance, consider the siteswap pattern 5,3,4. By performing a swap between the two adjacent sites $j = 2$ and $k = 3$, we obtain a new siteswap pattern 5,5,2. If we swap the same two sites again, we go back to our original pattern 5,3,4. This also works in the general case, which means that the site swap operation is its own inverse (mathematicians call this an *involution*).

3. ANALYZING PATTERNS

Unfortunately, the algebraic criterion from the previous section doesn't provide us with an actual juggling sequence. Thus we still need an algorithm which lets us analyze a given pattern, determine whether it is a valid siteswap pattern, and construct some representation of the resulting juggling sequence. The Lua function listed in Appendix A does this. It takes a pattern (a list a.k.a. Lua table of nonnegative integer values) as input, checks for all possible error conditions, and returns two Lua tables `pat` and `objs` as result (unless it runs into an error condition, in which case it raises an exception with a trace of the computation in the error message for diagnostic purposes).

There are a lot of corner cases in the algorithm, and we won't go into all the gory details here, but note that we first check how many objects (balls) we need (making sure that the average throw is an integer), then we go through the objects one by one, filling in each throw in the `pat` table as if we were drawing a juggling diagram. We finally do another pass of the constructed `objs` table to make sure that we fill in any missing `pat` entries, and add the zero entries. If we encounter any collisions during the construction, we bail out with an exception. (The algorithm employs Kikito's *inspect* (https://github.com/kikito/inspect.lua) for creating textual representations of intermediate table results for the traces, so that we also get some useful diagnostics.)

The end result consists of two tables. The `pat` table is a finite representation of the actual juggling pattern. It is indexed by beats (counting from 1). The length of this table (the number of beats) will always be an exact multiple of the length of the input pattern. Each non-empty entry is a pair `{i,k}` where `i` is an object (ball) index running from 1 to the average throw (an integer), and `k` is the throw, i.e., the number of beats object `i` stays in the air. An entry can also be `{0,0}` to indicate a beat where no ball is thrown, corresponding to a zero entry in the input pattern.

The second table `objs` is used for internal bookkeeping during execution of the algorithm, but also provides useful information about each object (ball), so we include it in the output. The `objs` table is indexed by numbers running from 1 to the number of objects (balls) used in the juggling sequence. For each object `i`, it contains a list with a complete cycle of all throws object `i` goes through during the juggling sequence. This is also known as an *orbit* in the mathematical theory of juggling.

For instance, let's consider the pattern 5,3,4. The average throw of 12/3 = 4 tells us that we need 4 balls to juggle this pattern. The resulting `pat` table gives a single cycle of the juggling sequence, which looks as follows:

```
{ 1, 5 } { 2, 3 } { 3, 4 }
{ 4, 5 } { 2, 3 } { 1, 4 }
{ 3, 5 } { 2, 3 } { 4, 4 }
```

Thus the sequence starts by throwing object 1 for a duration of 5 beats. In the second beat, object 2 gets thrown for 3 beats. In the third beat, we throw object 3 for 4 beats. The next three beats repeat the same pattern 5,3,4, but with the objects 4, 2, and 1, and finally beats 7-9 repeat the same pattern 5,3,4 again, but with objects 3, 2, and 4. This completes the cycle, and the entire sequence repeats when object 1 lands again in the next beat, making this a 9 beat sequence. Note that in general it will take several repetitions of the siteswap pattern to make the entire juggling sequence repeat; in this example, we needed 3 instances of the original pattern.

The `objs` table shows us the cycle of throws each individual object goes through while running through one cycle of the juggling sequence. In this example, it looks as follows:

```
{ 5, 4 } { 3 } { 4, 5 } { 5, 4 }
```

This readily tells us that object 2 always gets thrown for a duration of 3 beats, while all the other objects alternate between throws of 4 and 5 during each cycle of the juggling sequence. It's not really clear how that information might be used for musical purposes, but it's there if you need it. E.g., it might be used to draw a visual representation of the sequence such as a juggling diagram.

4. TURNING SITESWAP PATTERNS INTO MUSIC

Let us finally discuss how we can use the pattern data computed with the algorithm from the previous section to drive a rhythm machine. One might question whether this makes any sense at all. After all, juggling is primarily a visual experience, how are we supposed to turn this into something musically interesting? But our juggling sequences are repetitive and, as music psychologist Diana Deutsch once remarked, almost any sequence of sonic events becomes musical once you put it on repeat [6]. Also, as we have seen, the input pattern may be repeated several times with different objects (balls), which should give us some amount of interesting variation in the musical rendering.

Here is one way to turn the `pat` data output by our algorithm into MIDI. We have two numbers on each beat to work with: the object number i and the throw number k. It's not hard to imagine that we can map objects to MIDI note numbers and throws to velocities, giving us a single MIDI note to play for each beat. This data can then be fed into some kind of synthesizer or sampler device to hear the sequence. For instance, here is a simple scheme that works quite well in practice:

- Our MIDI setup consists of two velocity values minvel and maxvel in the range 0 to 127, and a list of MIDI note numbers $n_1, n_2, ...$ that we cycle through in order to assign a MIDI note to each object.
- Map object $i > 0$ to the MIDI note number n_i. If the number of objects exceeds the number of MIDI notes in the list, we can just wrap around to the beginning, cycling as many times through the note list as needed to map a note number to each object.

- Map a throw value of $k > 0$ to the velocity (k-minthrow)/(maxthrow-minthrow)*(maxvel-minvel), where minthrow and maxthrow are the minimum and maximum throw values > 0 in the pattern, respectively. This gives us a linear interpolation of the velocity corresponding to k ranging from minvel to maxvel. This value then gets rounded to the nearest integer.

For instance, consider the pattern 5,3,4 (with minthrow = 3 and maxthrow = 5) from above. Using minvel = 80, maxvel = 120, and the MIDI notes 35 42 47 48 as the MIDI setup, this will produce the following note output in the corresponding 9-beat loop:

beat	object, throw	MIDI note
1	1, 5	35 120
2	2, 3	42 80
3	3, 4	47 100
4	4, 5	48 120
5	2, 3	42 80
6	1, 4	35 100
7	3, 5	47 120
8	2, 3	42 80
9	4, 4	48 100

5. PD PATCH

We have implemented these ideas as a simple Pd patch, which can be found in the accompanying materials [7], cf. Fig. 2.

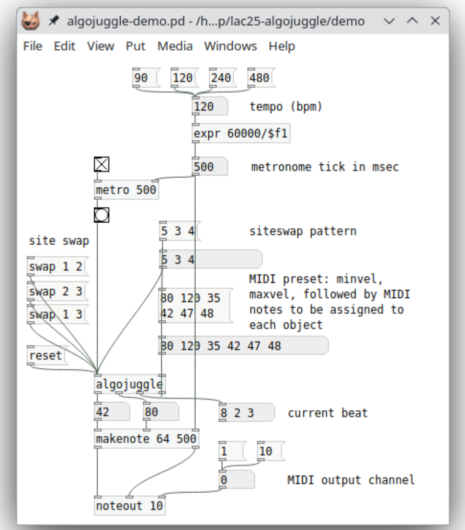

Figure 2: algojuggle demo patch.

The central component in this patch is the `algojuggle` object which maintains all the requisite data and cycles through the pattern with each `bang` message. The object is written in Lua and also incorporates the `analyze_pattern` algorithm from Appendix A. Note that in order to have this object recognized in Pd, you need to have the pd-lua loader extension installed [1] (please check the cited link for installation instructions).

The siteswap pattern is input as a list on the first inlet; the object then uses `analyze_pattern` to check that the input is a valid siteswap pattern and construct the actual juggling sequence. The MIDI setup, consisting of minvel, maxvel, and the list of note numbers as described in the previous section, gets passed on the second inlet. Upon receiving a `bang` message, the object outputs the MIDI note number and velocity of the next note in the pattern on the first two outlets, which get handed over to Pd's `makenote` to produce the stream of MIDI note ons and offs which are output with the `noteout` object on MIDI channel 10 (the MIDI drumkit channel) by default. To hear the notes, you will thus have to pipe Pd's MIDI output to some GM-compatible MIDI synthesizer such as qsynth.

For each `bang`, `algojuggle` also outputs the current position in the pattern along with the object number and throw value on the third outlet, so that you can watch the object cycle through the juggling sequence. It's conceivable that in the future this data might also be used to drive a real-time animation similar to what Jack Boyce's Juggling Lab provides online [5].

The `bang` messages are generated with Pd's `metro` object. This can be turned on and off with the toggle directly above the `metro` object, and the tempo can be controlled with the other GUI elements at the top of the patch, using either beats per minute (bpm) or millisecond (msec) values. Alternatively, you can also leave the `metro` object turned off, and step through the sequence manually by clicking the bang GUI object below the `metro` object, which can be useful for debugging purposes.

When loading the patch, the `algojuggle` object is already initialized with the data shown in the screenshot, i.e., 5 3 4 as the siteswap pattern and a MIDI setup with minvel = 80, maxvel = 120, and the MIDI notes 35 42 47 48 (in the General MIDI drumkit, these are the note numbers for "Acoustic Bass Drum", "Closed Hi-Hat", "Low Mid Tom", and "High Mid Tom", respectively). Thus you can just engage the `metro` toggle and the patch will start playing notes immediately. You can also try changing the output MIDI channel to 1 to generate a little melody instead, and adjust the notes to your liking.

Finally, the `algojuggle` object also implements the site swap operation described in Section 2. This can be used at any time, also during playback, which gives you a quick way to mutate the pattern. To these ends, the first inlet accepts a message of the form `swap i j`, where $i < j$ are two indices in the original siteswap pattern. These must be such that the distance $d = j - i$ is at most the throw value at position i, otherwise you will get an error message. If the pair constitutes a valid site swap, the resulting siteswap pattern is output on the fourth outlet, which in our patch is wired to the list GUI element for the input pattern which immediately feeds it back into `algojuggle`'s first inlet.

Note that to keep things simple, here we have only described the most basic version of the patch. Work has also started on an extended version of the patch (algojuggle-ex.pd) which includes preliminary MIDI controller support for live improvisation, MIDI clock sync to synchronize with DAW programs or external grooveboxes, and the possibility to run multiple instances of the extended patch in concert. Please check the README file in the repository [7] for details. The repository also includes some listening samples in both MIDI and audio (WAV) format.

6. CONCLUSION

After listening to the sequences produced with our patch, we hope that you can agree that it is possible to obtain interesting musical results using the method sketched out in this paper. The output can be quite surprising at times when experimenting with different input patterns, which means that our method can be used to supply ideas for compositional purposes. It might even be a good tool for live performances if you prepare a collection of interesting siteswap patterns beforehand.

On the other hand, this also means that it can be hard to control the kind of musical sequences that you get with this approach. You will have to experiment. Our implementation currently doesn't include a siteswap pattern generator, so you'd want to scour the web for lists of siteswap patterns (see, e.g., Allen Knutson's Siteswap FAQ [9]), or rely on existing siteswap generator and visualization programs such as Juggling Lab [4].

At this point it is still unclear whether it is possible to characterize the musical properties of sequences generated from siteswap patterns in a way that makes it easier to specifically design such patterns for musical purposes. On the positive side, this opens up opportunities for future research and musical experimentation. Here are some specific avenues for improving the algorithm and its implementation:

- Work on integrating a library or generator algorithm for musically interesting siteswap patterns. This will require an analysis of the musical results of the siteswap pattern approach and comparison with other generative methods.
- Enhance the MIDI sequence generation with techniques found in other beatmaking devices, such as loops, note repeats (ratchets), probabilistic variations, etc.
- Design a control scheme for expressive live improvisation, by interfacing to input controllers readily available for such purposes, such as DJ and pad controllers, sensor input, etc.

Acknowledgements

The authors would like to thank the anonymous reviewers for their helpful comments and suggestions.

7. REFERENCES

[1] pd-lua. https://agraef.github.io/pd-lua/.

[2] C. Barlow. *On Musiquantics*. Number 51 in Musikinformatik & Medientechnik. JGU, Mainz, 2012.

[3] T. Bovermann, J. Groten, A. de Campo, and G. Eckel. Juggling Sounds. In *Proceedings of the 2nd International Workshop on Interactive Sonification*, York, UK, 2007.

[4] J. Boyce. Juggling Lab. https://jugglinglab.org/.

[5] J. Boyce. Juggling Lab GIF server. https://jugglinglab.org/html/animinfo.html.

[6] D. Deutsch. Behaves So Strangely, Aug. 2010. https://www.radiolab.org/podcast/91513-behaves-so-strangely.

[7] A. Gräf. algojuggle-demo, Mar. 2025. https://github.com/agraef/algojuggle-demo.

[8] L. Harkleroad. *The Math Behind the Music*. Cambridge University Press, Cambridge, 2006.

[9] A. Knutson. Siteswap FAQ. http://www.juggling.org/help/siteswap/faq.html.

[10] B. Polster. Juggling Survey (The Mathematics of Juggling). https://www.qedcat.com/articles/juggling_survey.pdf.

[11] B. Polster. *The Mathematics of Juggling*. Springer, New York, NY, 2003.

[12] S. Renz. Algorithmische Rhythmen – Von Jonglierpatterns zu Klangsequenzen. Master's thesis, Johannes Gutenberg University, Mainz, 2024. https://gitlab.rlp.net/srenz/algojugglerhythm.

[13] G. Silveira. Using Juggling as a Controller for Computer-Generated Music: an Overview of the Creation of an Interactive System Between Juggling and Electronic Music. Bachelor's thesis, Universidade Federal de Pelotas, 2015.

[14] G. T. Toussaint. *The Geometry of Musical Rhythm: What Makes a "Good" Rhythm Good?* Chapman and Hall/CRC, Boca Raton London New York, 2nd edition, 2019.

[15] A. Wagenaar. Juggling as a controller of electronic music. Master's thesis, Conservatory of Amsterdam, 2009. https://www.arthurwagenaar.nl/wp-content/uploads/Juggling-as-a-controller-of-electronic-music.pdf.

[16] A. Willier and C. Marque. Juggling Gestures Analysis for Music Control. In I. Wachsmuth and T. Sowa, editors, *Gesture and Sign Language in Human-Computer Interaction*, pages 296–306, Berlin, Heidelberg, 2002. Springer.

A. LUA CODE

```lua
-- helper function to cycle through the
-- elements of a list
function cycle(list, i)
  i = (i-1) % #list + 1
  return list[i]
end

-- analyze a siteswap pattern
function analyze_pattern(pattern)
  local n = #pattern
  local sum = 0
  for _,k in ipairs(pattern) do
    sum = sum + k
  end
  local nobjs = sum/n
  -- nobjs must be integer, otherwise the
  -- pattern isn't valid
  if math.floor(nobjs) ~= nobjs then
    error("invalid pattern, average throw must
        be integer, but got " .. nobjs)
  end
  local objs = {}
  local pat = {}
  local obj = {} -- object for each position
  local start = {} -- actual start position
  local max_pat = 0 -- maximum index in pat
  local trace = "" -- execution trace
  for i = 1, nobjs do
    local last_i = i
    while obj[last_i] or cycle(pattern, last_i)
        <= 0 do
      -- look for the next free position
      last_i = last_i + 1
    end
    start[i] = last_i
    local last_k = cycle(pattern, last_i)
    local seq = { last_k }
    -- set of indices (mod n) already visited
    local i_set = { [last_i % n] = true }
    while true do
      trace = trace ..
        string.format("\n#%d: pos: %d (== %d),
            step: %d, seq: %s", i, last_i,
            (last_i-1) % n + 1, last_k,
            inspect(seq))
      if pat[last_i] then
        -- we already have a previous object at
        -- this position, bail out
        error(string.format("collision: #%d-#%d
            at pos %d%s", i, pat[last_i][1],
            last_i, trace))
      end
      if last_i > max_pat then
        max_pat = last_i
      end
      pat[last_i] = {i,last_k}
      obj[last_i] = i
      last_i = last_i+last_k
      last_k = cycle(pattern, last_i)
      if i_set[last_i % n] then
        -- position already visited, we're done
        -- but mark the position as occupied
        obj[last_i] = i
        break
      elseif #seq > n then
        -- if we come here, something is broken
        error("this can't happen, please check
            the algorithm!" .. trace)
      end
      table.insert(seq, last_k)
      i_set[last_i % n] = true
    end
    objs[i] = seq
  end
  -- go through the objs table once again, fill
  -- in missing pat entries
  if max_pat % n ~= 0 then
    -- make sure that we extend the pattern to a
    -- full cycle at the end
    max_pat = max_pat + n - max_pat % n
  end
  for i = 1, nobjs do
    local last_i = start[i]
    local j = 1
    local seq = objs[i]
    while last_i <= max_pat do
      local last_k = cycle(seq, j)
      if not pat[last_i] then
        pat[last_i] = {i,last_k}
        trace = trace ..
          string.format("\n#%d: pat[%d]: %s", i,
               last_i, inspect(pat[last_i]))
      elseif pat[last_i][1] ~= i then
        trace = trace ..
          string.format("\n#%d: pat[%d]: %s", i,
               last_i, inspect({i,last_k}))
        -- collision, bail out
        error(string.format("collision: #%d-#%d
            at pos %d%s", i, pat[last_i][1],
            last_i, trace))
      end
      last_i = last_i + last_k
      j = j+1
    end
  end
  -- fill in empty positions (0 = "no object")
  for i = 1, max_pat do
    if not pat[i] then
      pat[i] = {0,0}
    end
  end
  return pat, objs
end
```