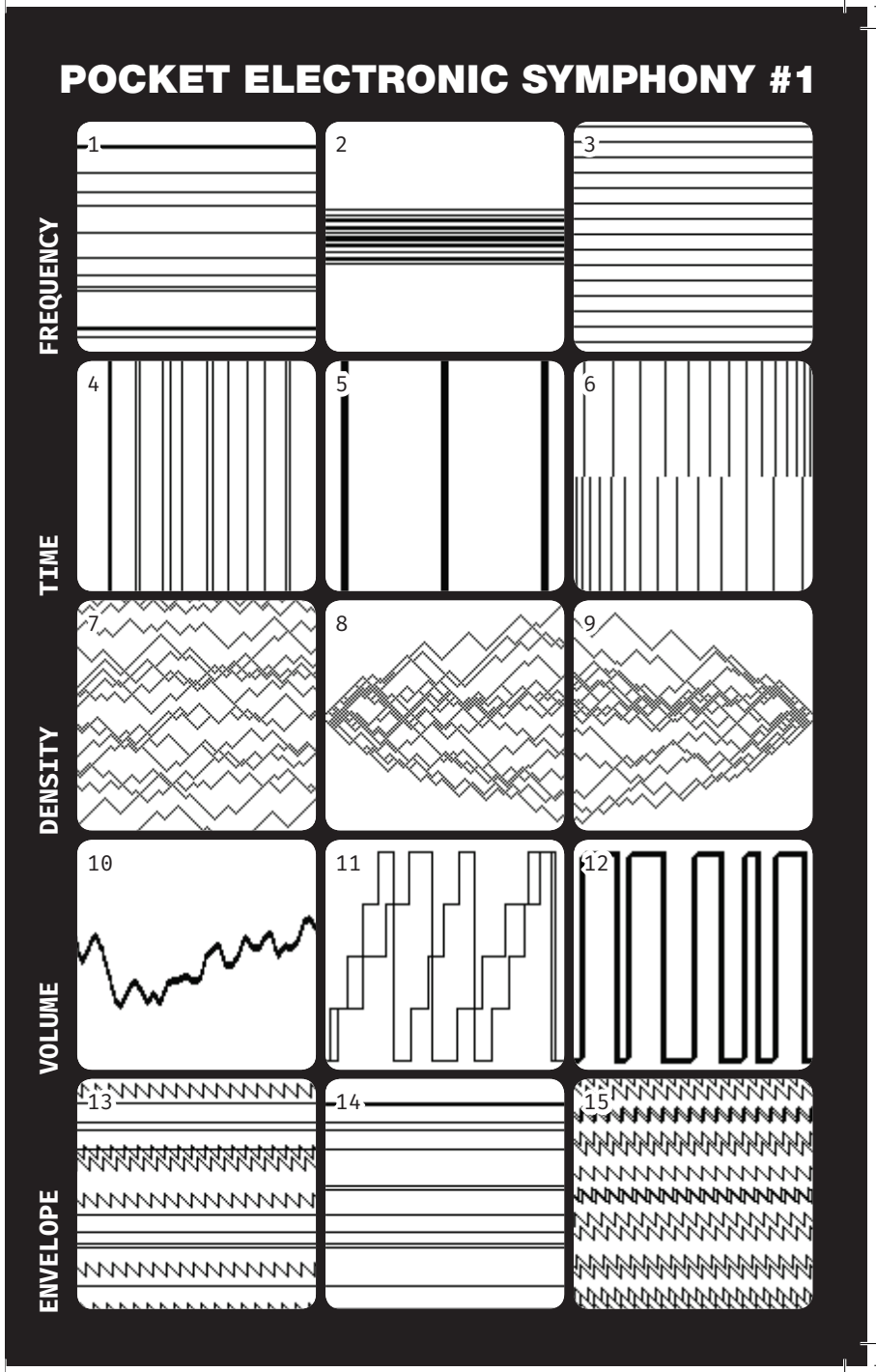
POCKET ELECTRONIC SYMPHONY #1



Pocket Electronic Symphony #1

Andreja Andric

for a solo performer with smartphone

Performing score,
JavaScript source code
& documentation

‡

// Code Comments

/* This script consists of a list of variables and of program code. The list of variables starts here. The three variables and the function next() form the linear congruent random number generator that the program uses to make micro-decisions while it generates the sound file */

/* During the development of the sound, certain changes take place, but not at every byte, because that would be too fast to follow. T is the value of a basic time interval (measured in bytes of the sound flow) in which mostly any change can happen. T equals 882 bytes, which equals 1/400 s at the rate of 44100 frames per second per stereo channel */

/* D is the current sound density, or how many individual notes the cluster currently contains. V indicates volume of each individual sound in the current cluster, measured in units of scale that constitute 16-bit sound, in other words, in units inside the interval between -32768 and 32767 */

/* Each event has its toggle that turns it on and off, as explained above. Each one has also its own timer and time interval measured in units of T above, in which it may

```
<script>

  var a = 2300446;
  var b = 1531103;
  var c = 2309401;

  function next() {
    a = (a * b % Number.MAX_SAFE_INTEGER + c) %
    ↠Number.MAX_SAFE_INTEGER;
    if (a < 0) a = -a; return a;
  }


  var T = 882;


  var D = 11;

  var V = 1888;


  var STATES = 15;
```

```
 appear. STATES is the overall number of toggles, and
     corresponing events and time intervals, which is
                                constant and equals 15.
   i is the counter of bytes in the sound signal (the
                 sound is 16bit stereo 44100frames/s).
 tx is the counter of time expressed in time measured
                         in intervals of T (1/400s) */


/* TTMIN and TTMAX respectively stand for minimum and
    maximum values between which the 15 intervals TT[]
 (explained below) will always find themselves. These
 maximum and minimum time values are not constant but
                       change when Toggle 4 is active.
  TT[] contains 15 periods (counting in counts of T's
 above) in which other changes take place, as already
                                            explained.
 ST[] contains 15 states that change according to the
 TT time intervals and which, when set to 1, indicate
  that the corresponding event will take place. These
states trigger changes to the sound flow which happen
     at intervals, as ST is 1 only during one sample.
    SST[] contains 15 states that keep the value of 1
 or 0 during the whole interval. These states trigger
      continuous kind of change which happen at every
            frame, while these states are at value 1.
      Slide is the amount of gradual pitch shift of a
   cluster (the change in pitch that goes in a single
                                            direction).
limit[] contains highest and lowest limits for V (see
   the previous set of cluster generation variables).
    A contains smallest interval that participates in
 building a cluster and at the same time the amount of
random change in pitch that go in opposite directions.
```

```
var i = 0;

var tx = 0;


var TTMIN = 100;
var TTMAX = 200;
var TT = [];
for (j = 0; j < STATES; j++)
   TT[j] = TTMIN + next() / 103 % (TTMAX - TTMIN);


var ST = [0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0];


var SST = [0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0];


var slide = 0;

var limit = [24000, 30000];

var A = 1;
```

```
      DMAX states maximum cluster density (set to 60).
        Part is a variable whose binary representation
   contains the state of all the toggles and therefore
       is used to control the flow of the composition.
    Only those changes where corresponding bit in Part
    variable is 1 will happen, and, since there are 15
   toggles, only lowest 15 bits count. For instance if
Part = 129, (0000000100000001 in binary representation)
   then only changes of type 0 and 7, corresponding to
                     toggles 1 and 8, will take place.
     IncreaseTime is an additional toggle that decides
     whether time intervals are increased or decreased
    all together on successive pushes of Toggle 6. The
    corresponding toggle changes to light blue or dark
    blue to indicate the difference. All other toggles
     change to dark blue only when active, and back to
    standard grey color when inactive. Toggle 6 is the
                               only 3-state toggle */
```

```
var DMAX = 60;
var part = 0;
```

```
var IncreaseTime = true;
```

```
 /* In this section we assign a listener function to
each toggle, so that variable part, described above,
     is modified accordingly with each press of each
    button. As mentioned above, Toggle 6 is the only
  3-state toggle, while all the others are two state
                                              togles.
 All toggles change color between dark blue (active)
and light grey (inactive), except the toggle 6 which
              also has a third color (light blue) */
```

```
var btnToggle = [0,0,0, 0,0,0, 0,0,0, 0,0,0, 0,0,0];

for(cc=0;cc<STATES;cc++) {
   btnToggle[cc]=document.
   ↠getElementById('btnToggle'+cc);
   let clist = btnToggle[cc].classList;

   if (cc == 5) {
      (function(index) {
         btnToggle[index].addEventListener(
         ↠"click", function() {
            part = part ^ 32;
            if (part & 32) {
               IncreaseTime = !IncreaseTime;
               if (IncreaseTime) {
                  clist.
                  ↠remove('darkblueButtonStyle');
                  clist.add('blueButtonStyle');
               } else {
                  clist.remove('blueButtonStyle');
                  clist.add('darkblueButtonStyle');
               }
            } else {
               clist.remove('blueButtonStyle');
               clist.remove('darkblueButtonStyle');
            }
            resetChangeTriggers();
         }, false);
      })(cc);
   } else {
      (function(index) {
         btnToggle[index].addEventListener(
         ↠"click", function() {
            var expon;
```

```
            (index == 0) ? expon = 1 : expon =
            →→Math.pow(2, index);
            part = part ^ expon;
            if (part & expon)
               clist.add('toggleButtonStyle');
            else
               clist.remove('toggleButtonStyle');
            resetChangeTriggers();
         }, false);
      })(cc);
   }
}


function resetChangeTriggers() {
   for (j = 0; j < STATES; j++) {
      ST[j] = 0; SST[j] = 0;
   }
}


var f = [];
var v = [];
f[0] = 3 + next() / 77 % 9;
for (j = 1; j < DMAX; j++)
   f[j] = f[j - 1] + A;
for (j = 0; j < DMAX; j++)
   v[j] = 0;
```

```
       /* Variables v[] and f[] define the individual
components of the sound cluster. f[]'s contain current
    increments of amplitudes of individual components.
       The faster an amplitude grows the more often it
      completes a cycle of changes between the highest
    and lowest amplitude limits, and therefore it is a
 measure of pitch. v[]'s are the current values of the
    amplitudes of individual components. The scale and
   effective pitch actually depends also on V, because
  the smaller the V, the faster an amplitude completes
  its cycle, so in a way, the triad f[] and V define a
   bidimensional musical scale in which all the voices
                                   of the cluster move.
```

```
    Saw indicates the amount of voices with saw-like
  envelope. Saw=1 indicates that 1/10th of all voices
    in the cluster will have saw-like envelope. Saw=5
   indicates that 1/2 have saw-like  envelope whereas
 the others have flat envelope. Saw=10 indicates that
   all voices have saw-like envelope. Saw=0 indicates
  that all voices have flate envelope. Regulating the
value of Saw variable is effected by means of Toggles
                                    13, 14 and 15 */
var Saw = 0;
```

```
       /* Variables aCtx, jsProcessor and gainNode,
      and function createAudioContext() are used for
 communicating with the sound card via Web Audio API.
   bufferSize is a constant that indicates the amount
  of data the sound card will receive at every cycle.
  leftBuffer and rightBuffer are memory buffers where
  the sound data will be stored for the sound card to
play in each cycle. isPlaying and controlsOn are both
  toggles that are either true or false. isPlaying is
used to control audio playback and controlsOn is used
 to switch between control interface and help screen.
 Function initAudio() makes necessary initializations
     for real time audio and sets copyBuffer() as the
   callback function that the sound card will call at
regular intervals to fetch the data inside leftBuffer
                                    and rightBuffer.
   initAudio() will be the first thing to do when the
                                        page loads */
function createAudioContext() {
    var contextClass = (window.AudioContext ||
        window.webkitAudioContext ||
        window.mozAudioContext ||
        window.oAudioContext);
    if (contextClass) {
        return new contextClass();
    } else {
        alert("Sorry. WebAudio API not supported. Try
        ↠using Google Chrome or Safari browser.");
        return null;
    }
}

var aCtx = createAudioContext();
var jsProcessor = 0;
var gainNode;
var bufferSize = 16384;
var leftBuffer = new Array(bufferSize);
var rightBuffer = new Array(bufferSize);
var isPlaying;
var controlsOn;
```

```
function initAudio() {
   if (typeof aCtx !== 'undefined') {
      if (/iPhone|iPad|iPod/i.test(
      ↠navigator.userAgent)) {
         var el = document.getElementById(
         ↠'overlayStartup');
         el.style.display = 'block';
      } else {
         jsProcessor = aCtx.createScriptProcessor
         ↠(bufferSize, 0, 2);
         jsProcessor.onaudioprocess = copyBuffer;

         gainNode = aCtx.createGain();
         jsProcessor.connect(gainNode);
         gainNode.connect(aCtx.destination);
         gainNode.gain.setTargetAtTime(
         ↠0, aCtx.currentTime, 0.015);

         document.getElementById('guiBtn').
         ↠addEventListener(
         ↠'click', toggleControls);
         document.getElementById('playBtn').
         ↠addEventListener('click', startStop);
         isPlaying = false;
         controlsOn = true;
      }
   } else {
      alert("Sorry. Audio context is not defined");
   }
}
```

```
    /* startStop toggles play and mute state of audio
                                            playback */
```

```
function startStop() {
   pBtn = document.getElementById(‘playBtn’);
   if (isPlaying == true) {
      isPlaying = false;
      pBtn.innerHTML = ‘> play’;
      gainNode.gain.setTargetAtTime(
      ↠0, aCtx.currentTime, 0.015);
   } else {
      isPlaying = true;
      pBtn.innerHTML = ‘x mute’;
      gainNode.gain.setTargetAtTime(
      ↠1.0, aCtx.currentTime, 0.015);
   }
}
```

```
   /* On the mobile phone, the performer sees one of
 two things: 1) the control interface, or 2) the help
 screen. Function toggleControls switches between the
two and rearranges the details on the visual controls
                                    to reflect this */
```

```
function toggleControls() {
   var instrEl = document.
   ↠getElementById(‘instrument’);
   var infoEl = document.getElementById(‘info’);
   var gBtn = document.getElementById(‘guiBtn’);
   var pBtn = document.getElementById(‘playBtn’);
   var headerEl = document.getElementById(‘header’);

   if (controlsOn == true) {
      controlsOn = false;
      gBtn.innerHTML = ‘controls’;
      instrEl.style.display = ‘none’;
      infoEl.style.display = ‘block’;
      gBtn.classList.add(‘lightBg’);
      pBtn.classList.add(‘lightBg’);
      headerEl.classList.add(‘lightBg’);
   } else {
      controlsOn = true;
```

```
      gBtn.innerHTML = ‘about’;
      instrEl.style.display = ‘block’;
      infoEl.style.display = ‘none’;
      gBtn.classList.remove(‘lightBg’);
      pBtn.classList.remove(‘lightBg’);
      headerEl.classList.remove(‘lightBg’);
   }
}


window.onload = initAudio();


function copyBuffer(event) {
   var leftChannel = event.outputBuffer.
   ↠getChannelData(0);
   var rightChannel = event.outputBuffer.
   ↠getChannelData(1);
   var n = leftChannel.length;

   for (var j = 0; j < n; j++) {
      leftChannel[j] = leftBuffer[j];
      rightChannel[j] = rightBuffer[j];
   }
   setTimeout(fillBuffer, 3);
}
```

```
   /* As stated above, the manner of sound generation
  consists in calculating the entire sound wave frame
by frame and sending it to the sound card in a memory
buffer at regular intervals. Calculation of one slice
of the waveform (which is 16384 frames long, as shown
    above, in constant bufferSize) happens inside the
                                  function fillBuffer.
     In addition, we use double buffering in order to
  optimise speed. The moment the sound card calls the
      callback function, the value of sound buffer is
   obtained by copying from another buffer, and then,
  while the soundcard is playing the current slice of
   the waveform, we construct the sound snippet (with
  the function fillBuffer) inside this second buffer,
   which will be copied to the sound card on the next
call to callback function copyBuffer. And the same is
                                  done in every cycle */
```

```
/* t is the frame count for the current slice of the
 waveform. j is a counter used in various occasions.
   aa and bb are, respectively, cumulative amplitude
 on the left and right speaker; in other words final
    values for each frame will be contained in these
variables before being placed in corresponding place
  in the memory buffer which will later be copied to
                                    the soundcard */

   /* Do the following until the buffer is filled */
      /* Check that we are on the limit of the basic
interval T when some events may happen. Update ST and
   SST according to the intervals TT as stated above.
      Take into account only those TT intervals where
           corresponding bit in part variable is 1 */

 /* In this section we handle each of the individual
               events. The events are the following:
    1. Recreate the cluster with random gaps between
                                               notes.

2. Recreate half of the cluster with no gaps between
                                    notes (in unison)
```

```
function fillBuffer() {

  var t = 0;
  var j = 0;

  var aa = 0;
  var bb = 0;


  while (t < bufferSize) {
     if (i % T == 0 || (i + 1) % T == 0 ||
        (i + 2) % T == 0 || (i + 3) % T == 0) {
        var p = part;
        for (k = 0; k < STATES; k++) {
           TT[k] = Math.floor(TT[k]);
           if (p % 2 == 1 && tx % TT[k] == 0) {
              ST[k] = 1;
              SST[k] = !SST[k];
           } else
              ST[k] = 0;
           p >>= 1;
        }
        if (ST[0] == 1) {
           var B = next() / 31 % 300;
           f[0] = B + next() / 77 % 9;
           for (j = 1; j < D; j++)
              f[j] = f[j-1] + next() / 441 % 9 + A;
        }
        if (ST[1] == 1) {
           var B = next() / 31 % 300;
           f[0] = B + next() / 77 % 9;
           for (j = 1; j < D / 2; j++)
              f[j] = f[j - 1];
        }
```

3. Recreate the cluster with smallest gaps between notes.

4. Reset the maximum and minimum time interval for Events. Reset time intervals for events to random values in between. Change the amount of pitch changes that go in the same direction.

5. Reset time intervals for events either to maximum and minimum alternately or to the mean.

6. Increase and decrease all time intervals towards maximum or minimum respectively on successive occasions. Change the amount of pitch changes in the same direction.

```
if (ST[2] == 1) {
   var B = next() / 31 % 300;
   f[0] = B + next() / 77 % 9;
   for (j = 1; j < D; j++)
      f[j] = f[j - 1] + 1;
}
if (ST[3] == 1) {
   TTMIN = 50 + next() / 77 % 500;
   TTMAX = TTMIN + next() / 77 % 500;
   for (j = 0; j < STATES; j++)
      TT[j] = TTMIN + next() / 103 % (
      ↠TTMAX - TTMIN);
   slide = -3 + next() / 55 % 6;
}
if (ST[4] == 1) {
   var ChooseMean = (
   ↠next() / 335 % 10 > 4 ? true : false);
   for (j = 0; j < STATES; j++) {
      if (ChooseMean)
         TT[j] = (TTMIN + TTMAX) / 2;
      else
         TT[j] = j%2 == 0 ? TTMIN : TTMAX;
   }
}
if (ST[5] == 1) {
   for (j = 0; j < STATES; j++) {
      if (IncreaseTime == true) {
         TT[j] = Math.floor(TT[j] * 1.2);
         if (TT[j] > 10 * TTMAX)
            TT[j] = TTMIN;
      }
      else {
         TT[j] = Math.floor(TT[j] * 0.8);
         if (TT[j] < 2)
```

7. Increase or decrease cluster density. Bounce back a little if more than maximum allowed or less than minimum allowed. Increase and decrease pitches of individual cluster components using the separate amounts of change for the change in the same direction and in opposite directions.

8. Increase cluster density. Reset to a small value if it becomes greater than the maximum allowed

9. Decrease cluster density. Reset to a large value if it becomes smaller than the minimum allowed.

10. Increase or decrease volume level. Bounce back a little if it becomes more than the maximum allowed or less than the minimum allowed. Change the amount of pitch changes that go in the opposite directions.

```
            TT[j] = TTMAX;
         }
      }
      slide = -3 + next() / 55 % 6;
   }
   if (SST[6] == 1) {
      var decision1 = next() / 33 % D;
      var decision2 = next() / 33 % D;
      var decision3 = next() / 33 % D;
      f[decision1] += A;
      f[decision2] -= A;
      f[decision3] += slide;
      D += (-3 + next() / 114 % 7);
      if (D <= 0)
         D = 5 + next() / 111 % 7;
      if (D > DMAX)
         D /= 2;
   }
   if (SST[7] == 1) {
      D += next() / 114 % 7;
      if (D > DMAX) D = 3;
   }
   if (SST[8] == 1) {
      D -= next() / 114 % 7;
      if (D <= 0)
         D = DMAX - next() / 111 % 7;
   }
   if (SST[9] == 1) {
      V += (-3 + next() / 112 % 7);
      if (V <= limit[0])
         V = limit[0] + next() / 111 % 700;
      if (V > limit[1])
         V /= 1.3;
      V = Math.floor(V);
```

```
                11. Change the volume limits stepwise.

12. Change the volume level to one of volume limits.

13. Apply a saw-like envelope to a random percentage
                                          of voices.
        14. Apply a flat envelope to all the voices.

  15. Apply a saw-like envelope to all the voices */
```

```
      A = -4 + next() / 131 % 9;
   }
   if (ST[10] == 1) {
      if (limit[0] <= 18000
         || limit[1] > 60000) {
         limit[0] = 40000 + 80 * (
         →→next() / 33 % 100);
         limit[1] = 50000 + 80 * (
         →→next() / 33 % 100);
      }
      else {
         limit[0] *= 0.675; limit[1] *= 0.7;
         limit[0] = Math.floor(limit[0]);
         limit[1] = Math.floor(limit[1]);
      }
   }
   if (SST[11] == 1) {
      if (next() / 333 % 10 > 5)
         V = limit[1];
      else
         V = limit[0];
   }
   if (ST[12] == 1) {
      Saw = next() / 543 % 10;
   }
   if (ST[13] == 1) {
      Saw = 0;
   }
   if (ST[14] == 1) {
      Saw = 10;
   }
   tx++;
}
```

```
   /* We construct the cluster. aa contains the left
part, bb the right part in the stereo sound picture,
   as stated above. Use Saw variable to determine if
    saw-like envelope is to be applied to any of the
                                            voices */
aa = 0;
bb = 0;

while (D >= DMAX)
   D /= 2;
if (D <= 0)
   D = 3;

var SawMade = false;
if (Saw == 0 || Saw == 10)
   SawMade = true;

for (j = 0; j < D; j++) {
   f[j] = Math.floor(f[j]);
   v[j] += f[j];
   if (v[j] > V)
      v[j] = -V;
   if (j % 2)
      aa += v[j];
   else
      bb += v[j];
   if (j > D * Saw / 10 && !SawMade) {
      aa *= ((TT[0] - (tx % TT[0])) / TT[0]);
      bb *= ((TT[0] - (tx % TT[0])) / TT[0]);
      SawMade = true;
   }
}

if (Saw == 10) {
   aa *= ((TT[0] - (tx % TT[0])) / TT[0]);
   bb *= ((TT[0] - (tx % TT[0])) / TT[0]);
}
```

```
            leftBuffer[t] = aa / 32768;
            i += 2;
            T = Math.floor(T);
            if (i % T == 0 || (i + 1) % T == 0) continue;
            rightBuffer[t] = bb / 32768;
            i += 2;
            t++;

        }

        D = Math.floor(D);

    }

// End of Code comments          </script>
```

Pocket Electronic Symphony is a smartphone symphony for a solo performer. The symphony is written as sound generating software which acts both as the score and the musical instrument. Using this software on their mobile phone, the performer changes the parameters of the sound generation process, navigates the successions of massive chords of synthesized sound and builds towering climaxes and suspenseful calm sections.

The work tries to create a new kind of symphonic sound for the mobile age and to find new space for passionate, dramatic and grand musical expression.

Pocket Electronic Symphony #1 by Andreja Andric
published on ‡ DobbeltDagger 2018
ISBN 978-87-970443-0-8
https://dobbeltdagger.net

Original concept and coding by Andreja Andric ‘17
Design and user experience coding by Anders Visti ‘18

The publication is released in the context of
DIEM Elektro concert Pocket Electronic Symphony
Feburary 22, ‘18 in Musikhuset Aarhus, Denmark.

Thanks go to wonderful Aarhus and Berlin musicians and artists who have expressed encouragement, appreciation and support: Eli Guðnason, Jakob Bangsø, Martin Lau, Olga Szymula, Kasper Lauritzen, Søren Krag, Jens T. Bertelsen, Merlyn Perez-Silva, Joachim D. S. Wölm and many others. Special thank you to Anders Visti for his generous effort on making this publication possible.

POCKET ELECTRONIC SYMPHONY #1

1. Recreate the cluster with random gaps between notes.
2. Recreate half of the cluster with no gaps between notes (in unison)
3. Recreate the cluster with smallest gaps between notes.
4. Reset the maximum and minimum time interval for events. Reset time intervals for events to random values in between. Change the amount of pitch changes that go in the same direction.
5. Reset time intervals for events either to maximum and minimum alternately or to the mean.
6. Increase and decrease all time intervals towards maximum or minimum respectively on successive occasions. Change the amount of pitch changes that go in the same direction.
7. Increase or decrease cluster density. Bounce back a little if more than maximum allowed or less than minimum allowed. Increase and decrease pitches of individual cluster components using the separate amounts of change for the change in the same direction and in opposite directions.
8. Increase cluster density. Reset to a small value if it becomes greater than the maximum allowed.
9. Decrease cluster density. Reset to a large value if it becomes smaller than the minimum allowed.
10. Increase or decrease volume level. Bounce back a little if it becomes greater than the maximum allowed or less than the minimum allowed. Change the amount of pitch changes that go in the opposite directions.
11. Change the volume limits stepwise.
12. Change the volume level to one of the volume limits.
13. Apply a saw-like envelope to a random percentage of voices.
14. Apply a flat envelope to all the voices.
15. Apply a saw-like envelope to all the voices.

Andreja Andric ♯ DobbeltDagger 2018 ISBN 978-87-970443-0-8C