The SysSon Platform

Technical Report TR-2016-10-1
Institute of Electronic Music and Acoustics, Graz
(Status: in progress)

Hanns Holger Rutz

October 2016

1 First Sonification Scenario

Following a project meeting on 16-Sep-2016, it was decided that the first scenario or template to work out is to look at QBO (quasi-biennial-oscillation) and ENSI (El Niño), using two sets of measured temperature data.[1]

We had worked with QBO sonification already in one of the previous workshops. One usually selects a specific range in the altitudes and rather equatorial coordinates (e.g. +/- 10 degrees). Longitudes are often averaged, but we will try to also look at longitudinal movements.

QBO and ENSO interact with each other in that strong amplitudes in ENSO (lower altitudes) are connected with the inverse phenomena in the QBO (higher altitudes). It would thus be interesting to be also able to sonically compare the two.

1.1 Data Files

- `5x5-climatology_2001-05-01_2016-05-01_RO_OPSv5.6.2_L2b_no_METOP_no_TerraSAR-X.nc`
  A high-resolution file with 5x5 latitude/longitude grid, 600 altitude levels, and over 180 months (10.1 GB)

- `5x30-climatology_2001-05-01_2016-05-01_RO_OPSv5.6.2_L2b_no_METOP_no_TerraSAR-X.nc`
  A slightly lower resolution file with 5x50 latitude/longitude grid (1.7 GB)

Initial problems:

- File selection filter did not know about HDF files (previously only NetCDF files were used); **Result: fixed**

- Heatmap plots somehow fail to gather the statistics of the data. **Result: It is just running very slowly (c. 12 minutes).**

- After stats finally complete, data is useless because we don't have fill value information, and apparently something different from NaN is used. **Result: Converted file to use NaNs**

- Date format not recognised. Time unit is "seconds since 1990-01-01T00:00:00", somehow that is not correctly parsed. **Result: fixed**

- The dimensions "Latitude" and "Longitude" are not correctly recognised, because the map-overlay option is not shown. **Result: fixed**

We might:

- look into updating the NetCDF library and drop Java 6 support. New artifact is called `netcdf4` and latest version is 4.6.6. This might improve performance. **Result: no speed improvement**

- introduce a simple table view, so one can quickly browse the data numerically.

- create a preference item to *disable* automatic removal of cache upon application quit.

- we also need to be able to adjust the cache size, because the 1.7 GB files produces already a 230 KB stats cache, therefore we will easily transcend the default cache size of 1 MB.

---

1. A different scenario discussed are "stratospheric sudden warmings", which happen around the poles in 30 km altitude. The sudden jumps in temperature of up to 70 degrees happen within weeks, and proceed from top to bottom.
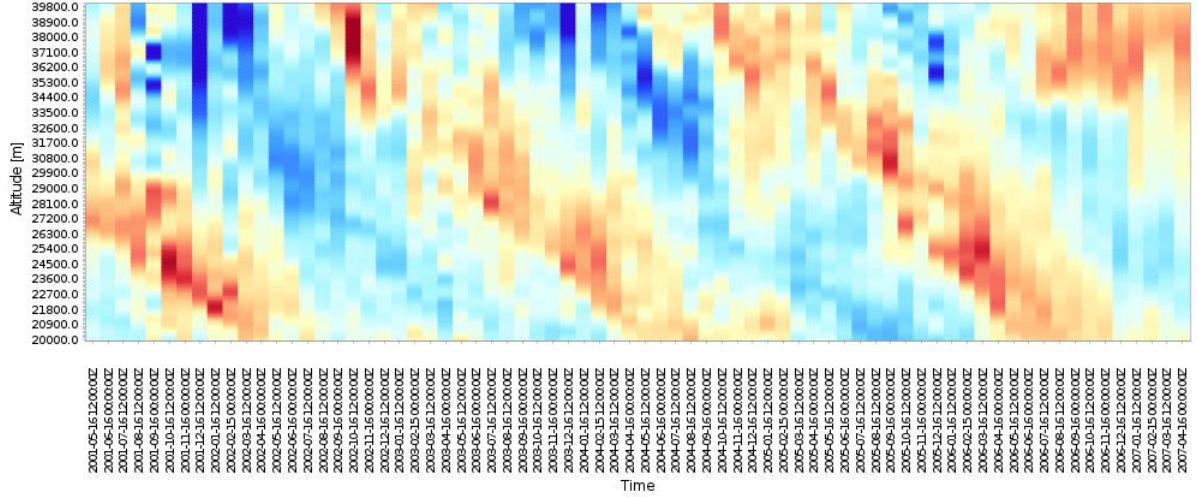
(a) Using monthly mean
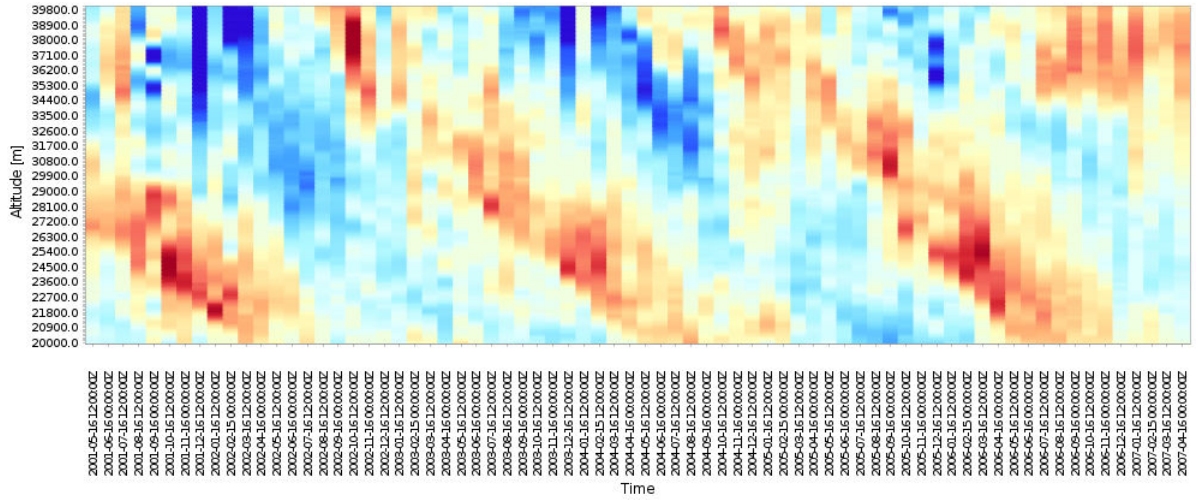
(b) Using monthly median

Figure 1: Temperature anomalies calculated with different norms. Indices where taken at longitude 105 degrees west, and latitude 2.5 degrees south. The expected periodicity of 28 to 29 months can be clearly seen.

After further inspection, fill values are correctly stored and found as -1e-10. Strangely stats show a max of 2.4e22 and a mean of 5.1e14. For example, for dry temperature those values appear in time index 90 and altitude indices 5 to 8. Here a "spike" of 4.49e23 in altitude index 5 declines towards index 8, and at index 9 the data is in the normal range again. We thus need to first preprocess that file and replace out-of-range values with the defined fill value. **Result: fixed by converting data.**

Intuition was, that because of possibly more slight glitches in the data, using the median instead of the mean for the calculation of anomalies might be better or more robust. We added this option to the anomaly utility of *SysSon*. Fig. 1 shows the difference between these two variants. It can be observed that using the mean monthly value as a basis results in a smoother plot, whereas using the median gives more pronounced deviations. Overall, the differences are small, though, and probably will not lead to significantly different sounds.

3

```
val vr     = Var("anom")
val dTime  = Dim(vr, "time")
val dAlt   = Dim(vr, "altitude")

val speed  = UserValue("speed", 1).kr
val tp     = dTime.play(speed)
val vp     = vr.play(tp, interp = 2)

val hot    = vp > 1
val amp    = hot * (vp.min(8) / 8)
val alt    = vp.axis(dAlt).values
val altMin = Reduce.min(alt)
val altMax = Reduce.max(alt)

val freq   = alt.linexp(altMin, altMax, 200, 4000)
val sin    = Mix.mono(SinOsc.ar(freq) * amp) / dAlt.size

output := Pan2.ar(sin)
```

Figure 2: First sketch; filtering only high temperatures, mapping them each to an oscillator.

1.2 First Sound Models

In the first model, time in the dataset is simply mapped to time in the sonification. We use a longitude and latitude index, and a slice of the altitudes, similar to the selection made in Fig. 1.

Questions:

- How do we "connect trajectories" and avoid steps (in time, because the altitude has a fine resolution)?

- How do we distinguish too-cold and too-hot? We could in any case project them onto left and right channel, respectively.

- Altitude resolution is high, we select roughly 200 bands.

The code for the first sketch is shown in Fig. 2. There is a simple threshold `vp > 1` that selects only the too-hot anomalies. Each altitude where this filter applies is represented by a sine oscillator with a frequency corresponding to the altitude (logarithmically scaled from 200 Hz to 4 kHz across the selected altitudes).

The sonic result is a dense texture with a metallic timbre. Not much details can be distinguished, but the overall downward glissando is clearly heard. One can also here some irregularities in the movement.

Looking again at the visual plot, it seems useful to reduce the amount of oscillators to capture only the significant properties, such as following the peak of the temperature profile. Without further offline pre-processing, the third-party UGen `ArrayMax` is a useful building block for detecting the global maximum in a multi-channel signal. In Fig. 3 this is used to produce the frequency of a single oscillator.

This works principally fine, however there are time slices where the global maximum jumps randomly between two local maxima. The next refinement thus would be to distinguish multiple trajectories and try to keep track of their respective frequencies such that jumps are avoided. A number of local maxima can be calculated, if `ArrayMax` is applied multiple times to a recursively filtered signal, in each iteration subtracting the last detected peak. This idea is illustrated in Fig. 4.

```
val vr    = Var("anom")
val dTime = Dim(vr, "time")
val dAlt  = Dim(vr, "altitude")

val speed = UserValue("speed", 1).kr
val tp    = dTime.play(speed)
val vp    = vr.play(tp, interp = 1)

val max   = ArrayMax.ar(vp)
val freq0 = max.index.linexp(0, dAlt.size - 1, 200, 4000)
val freq  = Ramp.ar(freq0, 1.0/speed)
val amp0  = max.value.clip(0, 8) / 8
val amp   = Ramp.ar(amp0, 1.0/speed)
val sin   = SinOsc.ar(freq) * amp

output := Seq(DC.ar(0), sin)
```

Figure 3: Second sketch; reducing to one oscillator at maximum bin.

```
...
val max       = ArrayMax.ar(vp)
val maxIdx    = max.index
val numAlt    = dAlt.size
val maskWidth = numAlt / 4
val mask      = vp * ChannelIndices(vp).absdif(maxIdx) > maskWidth
val max2      = ArrayMax.ar(mask)
val maxIdx2   = max2.index
val freq0     = maxIdx.linexp(0, numAlt - 1, 200, 4000)
val freq      = Ramp.ar(freq0, 1.0/speed)
val amp0      = max.value.clip(0, 8) / 8
val amp       = Ramp.ar(amp0, 1.0/speed)
val sin       = SinOsc.ar(freq) * amp

val freq02    = maxIdx2.linexp(0, numAlt - 1, 200, 4000)
val freq2     = Ramp.ar(freq02, 1.0/speed)
val amp02     = max2.value.clip(0, 8) / 8
val amp2      = Ramp.ar(amp02, 1.0/speed)
val sin2      = SinOsc.ar(freq2) * amp2
...
```

Figure 4: Using masking to find multiple "maxima"

1.3 Managing Voices and Trajectories

In order to manage multiple voices on the server side within one synth definition, the idea is to maintain a state matrix. Each row in the matrix corresponds to a "pre-allocated" voice. In order to allow voices to decay with an envelope when the trajectory ends, we need more voices than trajectories. A first approximation would be to allocate twice as many voices as the maximum number of expected trajectories. With the QBO data, we probably will want to track not more than four trajectories per polarity (e.g. two to four for the too-hot data, and two to four for the too-cold data). The matrix columns then carry the state of each voice:

- Whether it is currently enabled ("on-off").

- The last set frequency or trajectory magnitude

- The last set amplitude or trajectory strength

Without writing custom UGens, SuperCollider provides the UGens `LocalIn` and `LocalOut` to "store" state in a synthesis process. At the beginning of the graph, the previous state is read via `LocalIn`, at the end of the graph the updated state is written via `LocalOut`. This feedback mechanism incurs the costs of one control block latency for updated state data to appear earlier in the graph.

Fig. 5 shows the mechanism for tracing trajectories. Each voice here has only two state components, frequency and on-off. A first loop uses `ArrayMax` to find matches between the raw trajectory input data `freqIn` and `ampIn` and the current voice state `voiceFreq` and `voiceOnOff`. We can see how "logic" programming purely with UGens means to combine comparison signals (`sig_==` and `sig_!=`) with logical binary operators such as | (OR) and & (AND). At the end of the first loop, we have "re-activated" voices that continue an ongoing trajectory, and we have returned a multi-channel signal `noFounds` for input data that doesn't match voice data, i.e. data for which the distance in frequency is greater than a given threshold `maxDf`. A second loop then uses this information and assigns unused voices to these "new" trajectories. The special graph element `Trace` was used for debugging and is described further down.

Tracing UGens

During the course of the development of the voice management as outlined in the previous section, it became important to be able to precisely trace the values of several UGens over multiple audio frames or control blocks at the boundaries of crucial events. Using a simple built-in mechanism of SuperCollider such as the `Poll` UGen proved insufficient, because

- the ordering of poll prints depends on the final UGen graph topology

- comparing multiple successive values is difficult, because each poll is written to a new line and interleaved with other poll instances.

- the maximum resolution for triggers is Nyquist (for audio rate signals) or one per two control blocks (for control rate signals), because `Poll` uses a trigger input and not a gate.

In order to deal with this deficiency, and because the development of the voice structure was halted to due to bugs that were difficult to trace, a small library *ScalaCollider-Trace* was developed: `https://github.com/iem-projects/ScalaCollider-Trace`.

In its initial release, it has a simple mechanism for capturing a graph function, `traceGraph`, and for recording a trace via `playFor`. Traces are obtained by way of subjecting interesting

```scala
val numTraj     = 2  // number of trajectories followed
val numVoices   = numTraj * 2
val stateIn     = LocalIn.kr(Seq.fill(numVoices * 2)(0))
var voiceFreq   = Vector.tabulate(numVoices)(i => stateIn \ i): GE
var voiceOnOff  = Vector.tabulate(numVoices)(i => stateIn \ (i +
   numVoices)): GE
val voiceNos    = 0 until numVoices: GE
Trace(voiceFreq , "vc-freq-in")
Trace(voiceOnOff, "vc-on␣␣-in")
val freqIn      = "freq".kr(Vector.fill(numTraj)(0))
val ampIn       = "amp" .kr(Vector.fill(numTraj)(0))
Trace(freqIn, "in-freq")
Trace(ampIn , "in-amp")
val maxDf       = "max-df".kr(100f)
var activated   = Vector.fill(numVoices)(0: GE): GE

val noFounds = (0 until numTraj).map { tIdx =>
  val fIn       = freqIn \ tIdx
  val aIn       = ampIn  \ tIdx
  val isOn      = aIn > 0
  val freqMatch = (maxDf - (voiceFreq absdif fIn)).max(0)
  val bothOn    = voiceOnOff & isOn
  val bestIn    = 0 +: (freqMatch * (bothOn & !activated))
  val best      = ArrayMax.kr(bestIn)
  val bestIdx   = best.index - 1
  val bestMask  = voiceNos sig_== bestIdx
  activated     |= bestMask
  voiceFreq     = voiceFreq * !bestMask + fIn * bestMask
  Trace(bestIdx, s"f-match␣$tIdx")
  bestIdx sig_== -1
}

for (tIdx <- 0 until numTraj) {
  val fIn   = freqIn \ tIdx
  val aIn   = ampIn  \ tIdx
  val isOn  = aIn > 0
  val notFound  = noFounds(tIdx)
  val startTraj = notFound & isOn
  val free      = ArrayMax.kr(0 +: (startTraj & !activated))
  val freeIdx   = free.index - 1
  val freeMask  = voiceNos sig_== freeIdx
  activated     |= freeMask
  voiceFreq     = voiceFreq * !freeMask + fIn * freeMask
  Trace(freeIdx, s"f-free␣␣$tIdx")
}

voiceOnOff = activated  // release unused voices
Trace(voiceFreq , "vc-freq-out")
Trace(voiceOnOff, "vc-on␣␣-out")
val stateOut = Flatten(voiceFreq ++ voiceOnOff)
LocalOut.kr(stateOut)
```

Figure 5: First sketch for voice management

```
---------------------------
control-rate data: 8 frames
---------------------------
f-free  0       : -1.0    -1.0    -1.0    -1.0     0.0    -1.0     0.0    -1.0
f-free  1       : -1.0    -1.0    -1.0    -1.0    -1.0    -1.0    -1.0    -1.0
f-match 0       : -1.0    -1.0    -1.0    -1.0    -1.0     0.0    -1.0     0.0
f-match 1       : -1.0    -1.0    -1.0    -1.0    -1.0    -1.0    -1.0    -1.0
in-amp     [0]:   0.0     0.0     0.0     0.0     1.0     1.0     1.0     1.0
in-amp     [1]:   0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
in-freq    [0]:   0.0     0.0   300.0   300.0   300.0   300.0   500.0   500.0
in-freq    [1]:   0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
vc-freq-in [0]:   0.0     0.0     0.0     0.0     0.0   300.0   300.0   500.0
vc-freq-in [1]:   0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
vc-freq-in [2]:   0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
vc-freq-in [3]:   0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
vc-freq-out[0]:   0.0     0.0     0.0     0.0   300.0   300.0   500.0   500.0
vc-freq-out[1]:   0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
vc-freq-out[2]:   0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
vc-freq-out[3]:   0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
vc-on  -in [0]:   0.0     0.0     0.0     0.0     0.0     1.0     1.0     1.0
vc-on  -in [1]:   0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
vc-on  -in [2]:   0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
vc-on  -in [3]:   0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
vc-on  -out[0]:   0.0     0.0     0.0     0.0     1.0     1.0     1.0     1.0
vc-on  -out[1]:   0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
vc-on  -out[2]:   0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
vc-on  -out[3]:   0.0     0.0     0.0     0.0     0.0     0.0     0.0     0.0
```

Figure 6: Trace debug dump for eight control blocks, using the given sequence of values of *in-freq* and *in-amp* for the two trajectories. At the **fifth** step, the first trajectory's amplitude is set to greater than zero, resulting in an allocation of the first voice. In the **sixth** step, the frequency matching finds correspondence between first trajectory and first voice, thus preserving the first voice. In the **seventh**, the first trajectory's frequency jumps from 300 Hz to 500 Hz. In the first iteration, no matching trajectory for the first voice is found, making it available again, and thus the *f-free* indicates that it is picked again for the "new" trajectory. What we will eventually need is a *release* phase for that voice, before it can be allocated again.

UGens to a special `Trace` graph element. The result of a trace run is a future data matrix, and there is a simple pretty-print formatting that produces a tabular text view. The library is described in more detail on its project page.

The voice management we've come up with in Fig. 5 was developed using the tracing. The labels of the `Trace` elements can be directly found in the print-out that is shown in Fig. 6. The caption explains what is going on here and demonstrates the use of the trace mechanism to better understand the signal processing of a structure such as the voice management, which becomes arguably complex due to the way logical signals have to be represented in UGens.

Releasing Voices

In order to allow for voices to "decay", we need two signals, the original `voiceOnOff`, and one that delays the release (transition from one to zero). Options:

- `Slew` (with infinite rising and limited falling slope)

```
val stateInKr  = LocalIn.kr(Seq.fill(numVoices * 2)(0))
var voiceOnOff = ... // first channels in stateInKr

val voiceEnv   = Env.asr(attack = atk, release = rls)
val voiceEG    = EnvGen.ar(voiceEnv, gate = activated)
val voiceEGOn  = A2K.kr(voiceEG) sig_!= 0
voiceOnOff     = activated | voiceEGOn
val stateOutKr = ... // voiceOnOff plus other data
LocalOut.kr(stateOutKr)
```

Figure 7: Mechanism for releasing voices. The only other additional modification is the check for free voices which must now satisfy `def voiceAvail = !(activated | voiceOnOff)`.

- `EnvGen` – perhaps costly, but certainly the most flexible and general

- `TDelay` combined with `ToggleFF` or `SetResetFF`

- Setting `voiceOnOff` to the release duration (block count) and decrementing in the feedback loop one by one.

We will try and implement the `EnvGen` version. The problem here is that we can only calculate the correct `gate` parameter after the voice-matching loop, if we don't want to delay the attack. But we need to know the release state already during that loop, so we need another feedback mechanism, unless we decouple envelope generator and "held-gate". Detaching them sounds like a smart idea, because we can abstract over the particular envelope generation, and we have no issues with control-vs-audio rate. Then the last idea, counting down in the feedback loop, is perhaps the most straight forward approach to generate a correct dual gate/blocked signal, where

`def gate = voiceOnOff sig_== releaseBlocks`

On the other hand, an additional `LocalIn/Out.ar` does not incur much costs, and we can then check:

```
val envFB = LocalIn.ar(...)
val busy  = A2K.kr(envFB) > 0
```

This frees us from much counter arithmetic. It incurs the minimal costs of one extra busy cycle at the end when the envelope has actually already declined to zero.

Result: We simply re-used the existing logical control-rate signal `voiceOnOff` as shown in Fig. 7.

First Complete Sonification

The first complete implementation of the voice structure with the multiple-stage local-maxima search is shown in the following program:

```
implicit class MyGEOps(private val in: GE) /* extends AnyVal */ {
  def +: (head: Constant): GE = Flatten(Seq(head, in))
  def :+ (last: Constant): GE = Flatten(Seq(in, last))
  def ++ (that: GE      ): GE = Flatten(Seq(in, that))
  def \  (r: Range       ): GE = r.map(i => in \ i): GE
}

// ---- matrix ----

val vr    = Var("anom")
val dTime = Dim(vr, "time")
```

```
val dAlt  = Dim(vr, "altitude")

val speed = UserValue.kr("speed", 6)
val tp    = dTime.play(speed)
val vp    = vr.play(tp, interp = 1 /* 2 */)

// ---- configuration ----

// maximum number of trajectories followed
val numTraj      = 4

// maximum number of concurrently playing voices
val numVoices    = numTraj * 2

// maximum jump in altitude (normalized - full range equals one) before trajectory is interrupted
val maxDf        = UserValue.kr("traj-max-dif_[0-1]", 0.25)

// trajectory amplitude envelope attack duration
val egAtk        = UserValue.kr("traj-atk_[s]", 0.2)

// trajectory amplitude envelope release duration
val egRls        = UserValue.kr("traj-rls_[s]", 0.5)

// trajectory frequency and amplitude smear time
val lagTime      = 1.0

// threshold above which temperatures are considered anormal
val magThresh    = UserValue.kr("mag-thresh_[deg]", 1.5)

// maximum magnitude considered for anormal values
val magMax       = UserValue.kr("mag-max_[deg]", 8.0)

// sonification minimum oscillator frequency
val minFreq      = UserValue.kr("min-freq_[Hz]", 200)

// sonification maximum oscillator frequency
val maxFreq      = UserValue.kr("max-freq_[Hz]", 4000)

// ---- state ----

val stateInKr    = LocalIn.kr(Seq.fill(numVoices * 3 * 2)(0))

val voiceNos     = 0 until numVoices: GE
val numAlt       = dAlt.size
val maskWidth    = numAlt / 2 // 4
val vpChanIdx    = ChannelIndices(vp)

def mkSide(isUp: Boolean): (GE, GE) = {
  // ---- state ----
  val stateOff     = if (isUp) 0 else 3
  var voiceFreq    = stateInKr \ ((numVoices * (stateOff+0)) until (numVoices * (stateOff+1)))
  var voiceAmp     = stateInKr \ ((numVoices * (stateOff+1)) until (numVoices * (stateOff+2)))
  var voiceOnOff   = stateInKr \ ((numVoices * (stateOff+2)) until (numVoices * (stateOff+3)))

  // ---- trace trajectories ----

  def extract(in: GE, res: Seq[(GE, GE)], trjIdx: Int): Seq[(GE, GE)] =
    if (trjIdx == numTraj) res else {
      val (bestIdx, bestVal) =
        if (isUp) {
          val best = ArrayMax.kr(in)
          best.index -> best.value
        } else {
          val best = ArrayMin.kr(in)
          best.index -> best.value
        }

      val freq0     = bestIdx / (numAlt - 1) // .linexp(0, numAlt - 1, 200, 4000)
      val amp0      = if (isUp)
        bestVal.clip(magThresh, magMax).linlin(magThresh, magMax, 0, 1)
      else
        bestVal.clip(-magMax, -magThresh).linlin(-magMax, -magThresh, 1, 0)

      val mask      = in * vpChanIdx.absdif(bestIdx) > maskWidth
      extract(in = mask, res = res :+ (freq0 -> amp0), trjIdx = trjIdx + 1)
    }

  var activated    = Vector.fill(numVoices)(0: GE): GE

  val (freqInSq, ampInSq) = extract(in = A2K.kr(vp), res = Vector.empty, trjIdx = 0).unzip
  val freqIn = freqInSq: GE
  val ampIn  = ampInSq : GE

  // for each frequency, find the best past match
  val noFounds = (0 until numTraj).map { tIdx =>
    val fIn        = freqIn \ tIdx
    val aIn        = ampIn  \ tIdx
    val isOn       = aIn > 0

    val freqMatch  = (maxDf - (voiceFreq absdif fIn)).max(0)
    val bothOn     = voiceOnOff & isOn
    val bestIn     = 0 +: (freqMatch * (bothOn & !activated))
    val best       = ArrayMax.kr(bestIn)
    val bestIdx    = best.index - 1
```

```
   val bestMask    = voiceNos sig_== bestIdx
   activated       |= bestMask
   val bestMaskN   = !bestMask
   voiceFreq       = voiceFreq * bestMaskN + fIn * bestMask
   voiceAmp        = voiceAmp  * bestMaskN + aIn * bestMask

   bestIdx sig_== -1
 }

 for (tIdx <- 0 until numTraj) {
   val fIn           = freqIn \ tIdx
   val aIn           = ampIn  \ tIdx
   val isOn          = aIn > 0
   val voiceAvail    = !(activated | voiceOnOff)

   val notFound      = noFounds(tIdx)
   val startTraj     = notFound & isOn
   val free          = ArrayMax.kr(0 +: (startTraj & voiceAvail))
   val freeIdx       = free.index - 1
   val freeMask      = voiceNos sig_== freeIdx
   activated         |= freeMask
   val freeMaskN     = !freeMask
   voiceFreq         = voiceFreq * freeMaskN + fIn * freeMask
   voiceAmp          = voiceAmp  * freeMaskN + aIn * freeMask
 }

 // ---- voice generation ----
 val voiceEnv     = Env.asr(attack = egAtk, release = egRls)
 val voiceEG      = EnvGen.ar(voiceEnv, gate = activated)

 // ---- state out ----
 val voiceEGOn = A2K.kr(voiceEG) sig_!= 0
 voiceOnOff    = activated | voiceEGOn

 val stateOutKr  = voiceFreq ++ voiceAmp ++ voiceOnOff

 // ---- sound generation ----

 // gate so attack doesn't lag
 val lagTimeGt = (activated sig_== Delay1.kr(activated)) * lagTime
 val ampScale  = voiceAmp.linexp(0, 1, -20.dbamp, 0.dbamp)
 val ampLag    = Lag.ar(voiceAmp , time = lagTimeGt)
 val freqScale = voiceFreq.linexp(0, 1, minFreq, maxFreq)
 val freqLag   = Lag.ar(freqScale, time = lagTimeGt)
 val sines     = SinOsc.ar(freqLag) * voiceEG * ampLag
 val mix       = Mix(sines)
 (mix -> stateOutKr)
}

val (left , leftState)  = mkSide(true )
val (right, rightState) = mkSide(false)

LocalOut.kr(leftState ++ rightState)

val mix   = Limiter.ar(Seq(/* DC.ar(0) */ left, right))

output := mix
```

The settings for the test data set are shown in Fig. 8. A bounce of the audio output was made, a sonogram of which is shown in Fig. 9.

Discussion: The perceived sound strongly depends on the parameters, the trajectory coherence fraction `maxDf`, the oscillator frequency range `minFreq` and `maxFreq`, and the envelope release time `egRls`. We are projecting the positive anomalies to the left channel, and the negative anomalies to the right channel. While the glissandi can be heard individually on both channels, the rhythmic regularity and shift against each other is still difficult to perceive. Possible routes from here are the use of different frequency registers (as was done in the SysSon workshop), or the decoupling of frequency and altitude altogether.

1.4 Working with Timbre
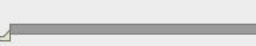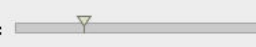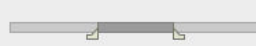
In the next step, we introduce two different frequency ranges for cold and hot anomalies, and further try to distinguish them perceptually by changing the type of oscillator (Fig. 10). For the hot anomalies a lower register (by default) is chosen, and the oscillator is a distorted Brownian noise fed through a resonator and mixed with a sawtooth oscillator one octave below that is fed through a low-pass filter. For the cold anomalies, a dust generator is fed through a resonator. Furthermore, an auxiliary signal with user-definable amplitude adds information about the time grid, playing

Figure 8: Matrix selection and parameter settings for sonification-3.

Figure 9: Sonogram from the bounce of sonification-3. Time elapses from left-to-right (duration 30 seconds), frequencies from bottom-to-top (lowest excited frequency is 200 Hz, highest excited frequency is 4 kHz. Top half shows left channel, corresponding to too-hot anomalies. Bottom half shows right channel, corresponding to too-cold anomalies.

short filtered noise pulses every month (for frequencies up to 6 months per second) and accent on Januaries and Julies (Fig. 11). **Assessment:** We should have UGen support for correctly using the time dimension in terms of calendar decomposition.

Fig. 12 shows a comparison of the anomalies plot and the sonogram of the bounced sonification. While one can see the correlation, especially the cold anomalies are not very well separated as trajectories, leading to more discontinuities than perhaps necessary.
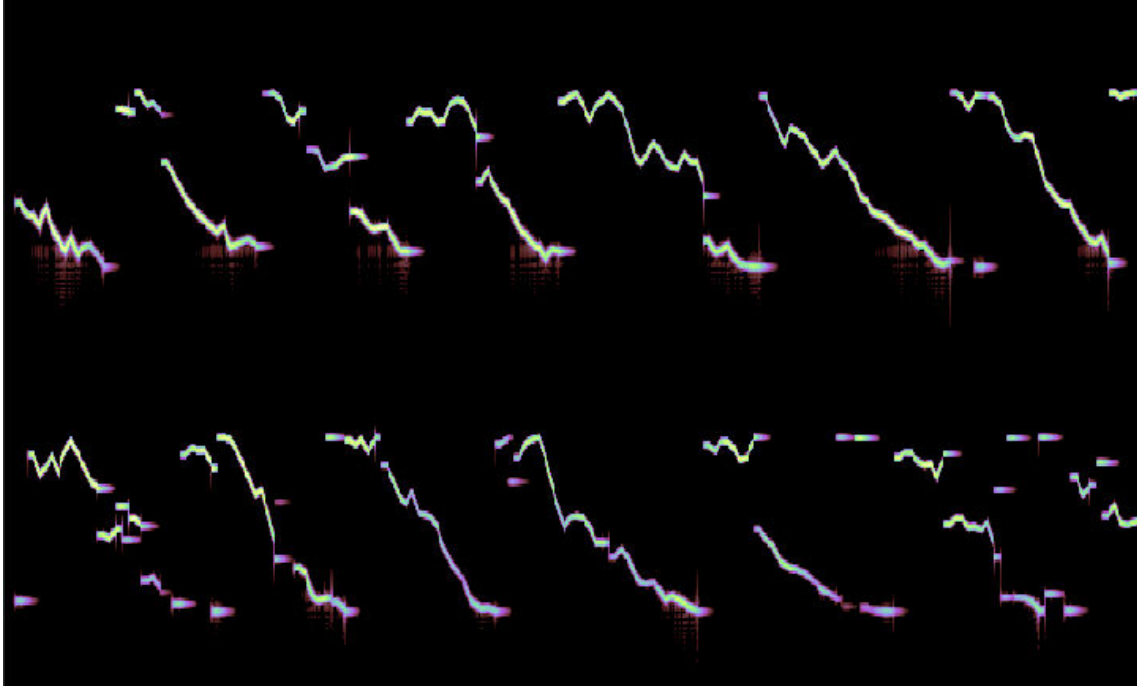
We went back to compare the anomalies based on monthly median versus mean value. The result is shown in Fig. 13. It seems that although the anomalies are slightly stronger when using the median, the trajectory following works better with the mean values, which can be observed especially towards the end of the sonogram in the cold anomalies (right channel or shown on the bottom half). The bounce for the mean based sonification is uploaded here: `https://soundcloud.com/syssonproject/sonif-4-based-on-median-anomalies`

Variants

In order to observe the rhythmic structure, it might be sufficient to actually listen only to a single altitude level, or to two altitude levels spaced apart. This was done in the following two cases:

- 30 km only: https://soundcloud.com/syssonproject/sonif-4-single-altitude

- 24 km vs 36 km: https://soundcloud.com/syssonproject/sonif-4-dual-altitude

```
val filterQ   = UserValue.kr("filter_Q_(1_to_100)", 20)
val filterRQ  = filterQ.reciprocal

val minFreq   = if (isUp) minFreqH else minFreqC
val maxFreq   = if (isUp) maxFreqH else maxFreqC
val freqScale = voiceFreq.linexp(0, 1, minFreq, maxFreq)
val freqLag   = Lag.ar(freqScale, time = lagTimeGt)
val osc       = if (isUp) {
    LPF.ar(Saw.ar(freqLag/2), freqLag) * 0.25 +
    Resonz.ar(BrownNoise.ar.sqrt, freqLag, rq = filterRQ) * 1.5
  } else {
    Resonz.ar(Dust2.ar(400), freqLag, rq = filterRQ) * 10
  }
val sines     = osc * voiceEG * ampLag
```

Figure 10: Different timbres and registers

```
// time grid indicator volume
val gridAmp       = UserValue.kr("time_grid_[dB]", -18).dbamp

val seconds       = tp
val daysPerYear   = 365.2422  // average according to NASA
val daysPerMonth  = daysPerYear / 12
val secsPerMonth  = daysPerMonth * 24 * 60 * 60
val months        = seconds / secsPerMonth
val month         = months.floor % 12
val isJan         = month sig_== 0
val isJul         = month sig_== 6

val monthPulse  = Impulse.ar(speed)
val gridDecTime = speed.reciprocal.min(0.5)
val gridDecay   = Decay.ar(monthPulse, gridDecTime)
val gridBase    = WhiteNoise.ar(gridDecay) * gridAmp
val gridJan     = Resonz.ar(gridBase, 1000, rq = 1.0)
val gridJul     = Resonz.ar(gridBase, 3500, rq = 0.5) * 1.5
val gridPlain0  = Resonz.ar(gridBase, 2000, rq = 2.0) * 0.25
val gridPlain   = Select.ar(speed <= 6, Seq(DC.ar(0), gridPlain0))
val gridIdx     = isJan | (isJul << 1)
val gridSig     = Select.ar(index = gridIdx, in = Seq(gridPlain,
    gridJan, gridJul))
Pan2.ar(gridSig)
```

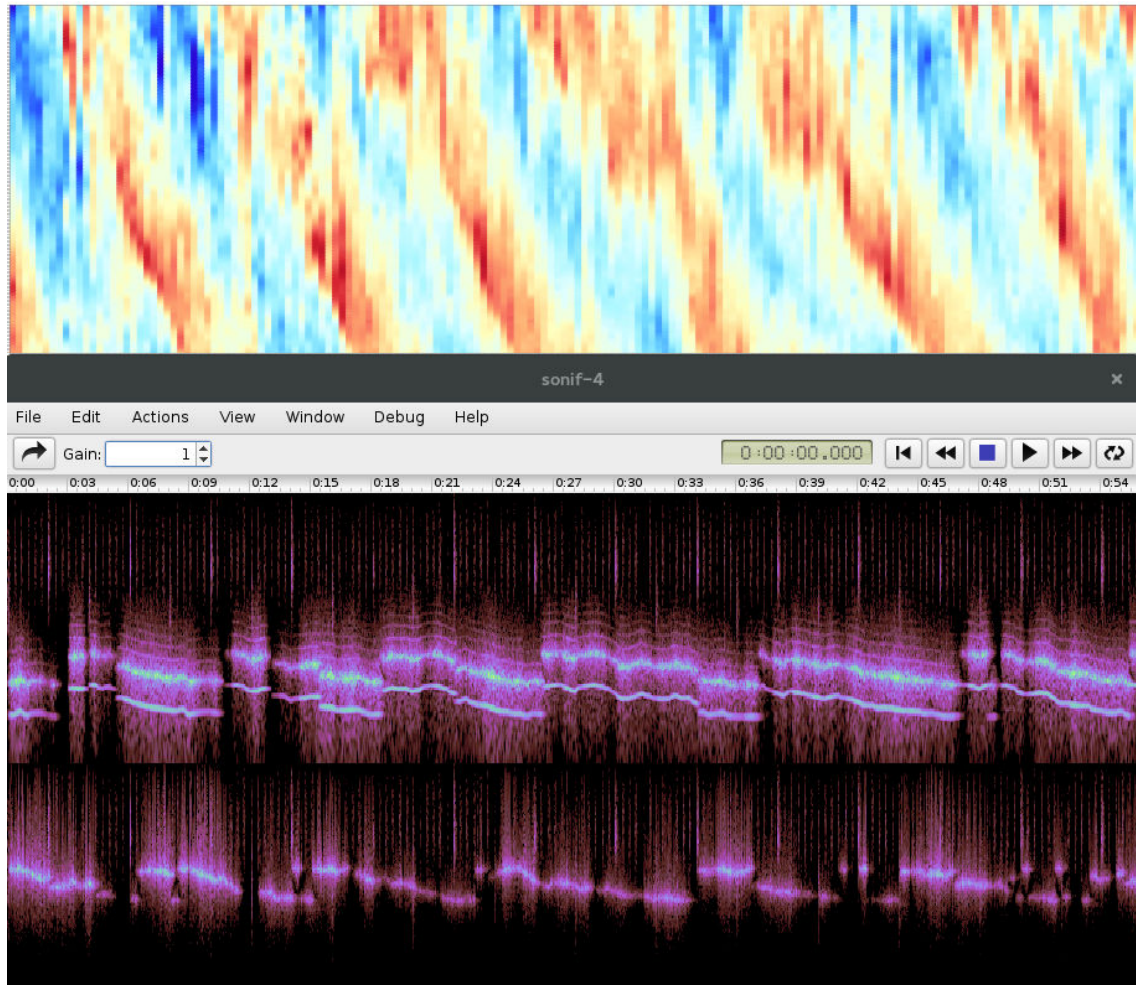Figure 11: Temporal grid marker sound generator
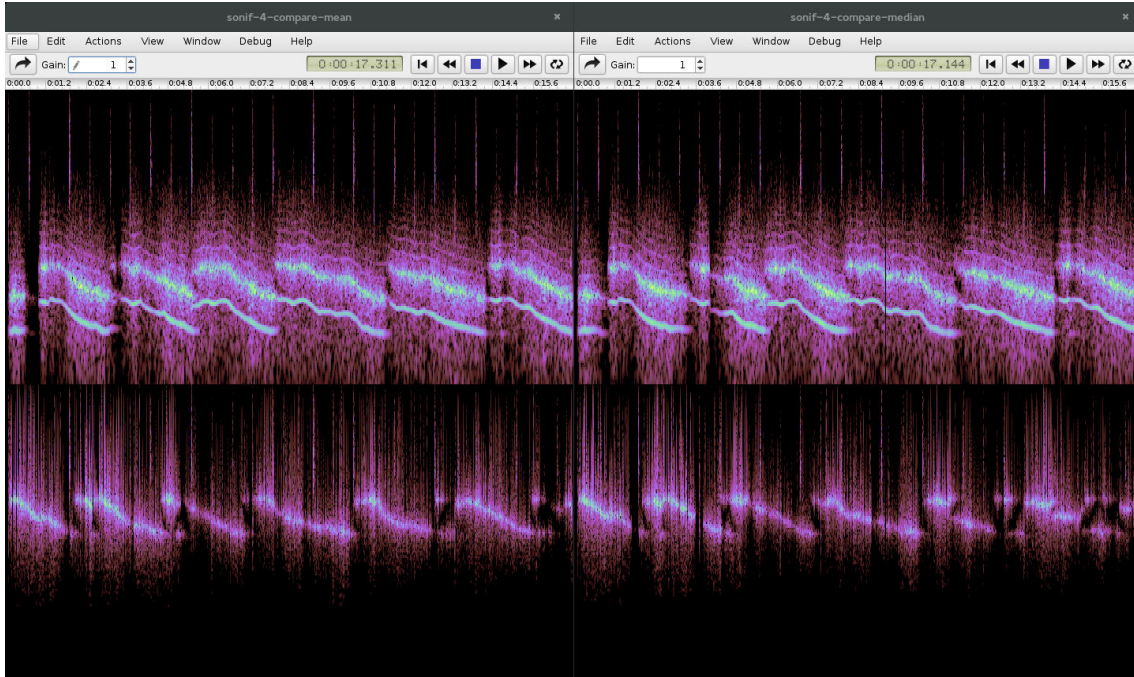
Figure 12: Sonification 4.

Figure 13: Sonification 4 using median based anomalies (left) vs. mean based anomalies (right).

2 Centroid

With the reduction to the local maximum, we somehow lose information about the breadth of the anomalies. Just picking the maximum does not take into account the centre of gravity or centroid of the band surrounding it. The next idea is thus to create a weighted sum of the energies around the maximum. To demonstrate what we mean, let's take the following simplified time slice vector (cf. `code/Centroid.scala` in the experiments project):

```
val ys = Vector(0.17, 0.14, 0.35, 0.25, -0.11, -0.43, -0.38, 0.08,
    0.64, 0.84, 1.00, 0.84, 0.64, 0.08, -0.38, -0.43, -0.11, 0.25,
    0.35, 0.14, -0.17)
ys.plot(discrete = true)
```

The plot is shown in Fig. 14. Then an intuitive way of determining the centroid would be:

```
val bestVal   = ys.max                 // 1.0
val bestIdx   = ys.indexOf(bestVal)    // 10
val thresh    = 0.2
val yi        = ys.indices

val beforeIdx = bestIdx - ys.take(bestIdx).reverse.indexWhere(_ <
    thresh) // 8
val afterIdx  = bestIdx + ys.drop(bestIdx + 1).indexWhere(_ < thresh)
    // 12
val mask      = yi.map(i => if (i >= beforeIdx && i <= afterIdx) 1
    else 0)
val inMask    = (ys zip mask).map { case (y, m) => y * m }
val inSum     = inMask.sum // 3.96
val wIdx      = (inMask.zipWithIndex).map { case (y, i) => y * i }
    .sum / inSum // 10
```
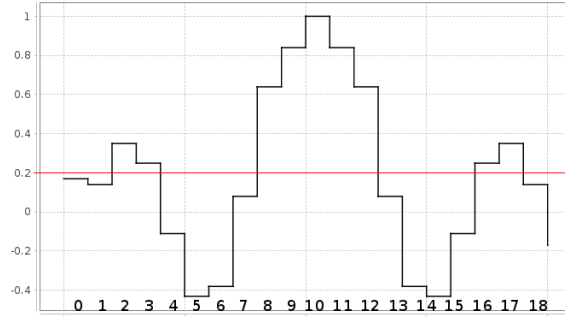
Figure 14: Test vector of 18 elements for centroid measurement. Threshold is 0.2

The problem is that we cannot use collections operations such as `take` and `drop` on the server, as the number of channels (collection size) depends on the input matrix. Also higher-order functions such as `indexWhere` are not available. We have to build everything around the `ArrayMin` and `ArrayMax` UGens again. Fig. 15 shows how this is done. Here, `isUp` is `true` for positive (hot) anomalies, and it is `false` for negative (cold) anomalies. To find the span, we calculate the reciprocal of the difference of the indices to the best-index (index of local maximum, or local minimum respectively). This we multiply with `isInvalid`, a logical signal that is high for the channels that are below the threshold and low for those above the threshold. Then using `ArrayMax` we can find the index closest to the best-index where the signal falls below the threshold. To allow cases where the signal does not fall below the threshold on either side, we are prepending the dummy zero channel (`0 +: x`), and need to take care to interpret a resulting zero index accordingly. The `Reduce.+(x)` elements sum up the channels of a signal. By multiplying with the channel indices and dividing by the total sum, we obtain the weighted index.

The result of this modified sonification is shown in Fig. 16. It can be seen that the weighted index version produces noticeably smoother and more steady glissandi. On the downside (?), we occasionally find short lived secondary trajectories close to the main trajectory. A fast and slow version of this new sonification are found here:

- `https://soundcloud.com/syssonproject/sonif-5`

- `https://soundcloud.com/syssonproject/sonif-5-long`

3   Blobs

There is "two dimensions" of value smear here, one is across altitudes, one is across time. So far we have only looked at the altitudes independently of neighbouring time slices. If we look at the plot again, left-hand side of Fig. 17, we perceive the visual gestalt certainly as two-dimensional. An idea would be to use two-dimensional blob detection, an example of which is shown in the right-hand side of Fig. 17. This picture was made using a blob detection library for Processing, apparently based on meta-balls: `https://www.niksula.hut.fi/~hkankaan/Homepages/metaballs.html` One could then use the blob data to "shape" the sound objects, calculate regression lines, etc.

3.1   Formalisation

The analysis is probably impossible, certainly tedious in the synth graph, so we need a mechanism to perform it offline beforehand.

```
def calcIndex(ys: GE, magMax: GE, isUp: Boolean): (GE, GE) = {
  val (bestVal, bestIdx) = if (isUp) {
    val best = ArrayMax.kr(ys)
    (best.value -> best.index)
  } else {
    val best = ArrayMin.kr(ys)
    (best.value -> best.index)
  }
  val yi        = ChannelIndices(ys)
  val isInvalid = if (isUp) ys < thresh else ys > -thresh
  val bestDist  = (bestIdx - yi).reciprocal * isInvalid
  val before    = ArrayMax.kr(0 +: bestDist)
  val beforeIdx = before.index
  val after     = ArrayMax.kr(0 +: -bestDist)
  val afterIdx  = (after.index - 1).wrap(0, NumChannels(ys) + 1)
  val mask      = yi >= beforeIdx & yi < afterIdx
  val inMask    = if (isUp) ys * mask else -ys * mask
  val inSum0    = Reduce.+(inMask)
  val inSum     = inSum0 + (inSum0 sig_== 0) // avoid division by zero
  val wIdx      = Reduce.+(inMask * yi) / inSum
  val wMag      = (inSum * allValid).clip(thresh, magMax)
    .linlin(thresh, magMax, 0, 1)
  (wIdx, wMag)
}
```

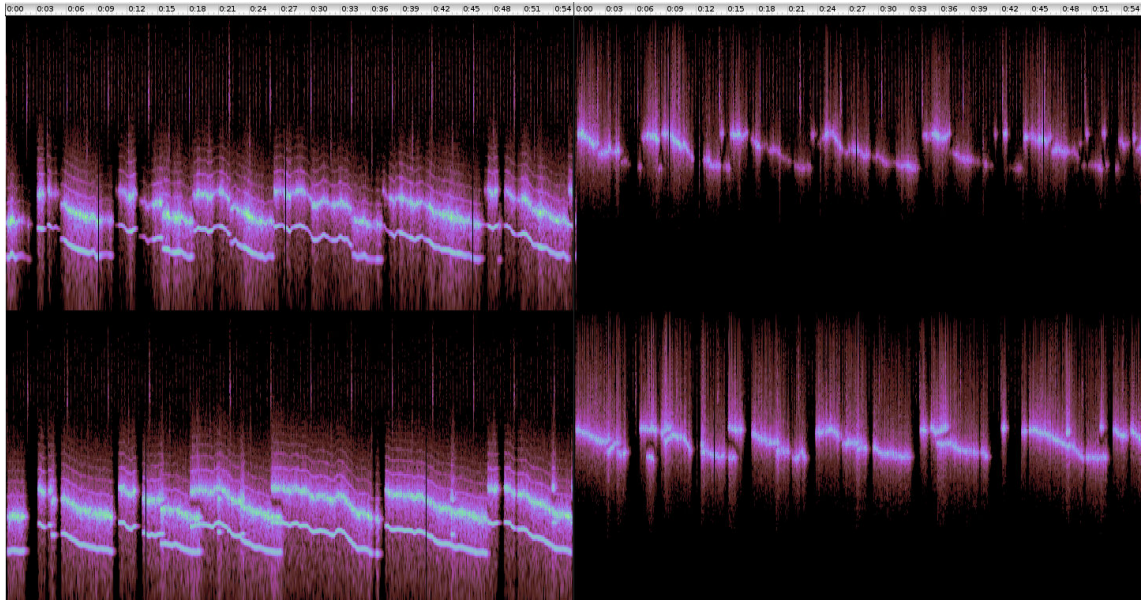Figure 15: Function that calculates the centroid of an input synthesis signal.

Figure 16: Comparison of sonification-4 – local-max only, top half – and sonification-5 – using centroid or weighted index, bottom half. Left side is positive anomalies, right side is negative anomalies.
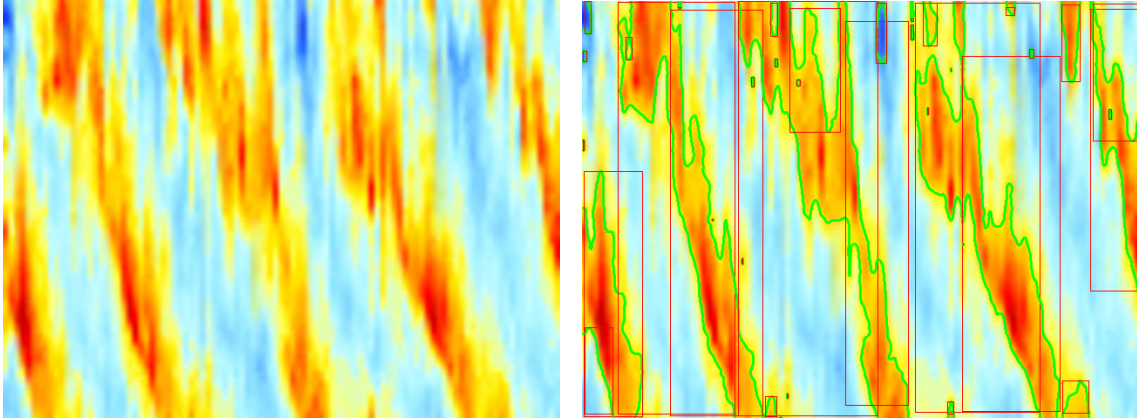
Figure 17: Thinking of the anomalies as diagonal "hoses"; left-hand side blurry plot, right-hand side with blobs detected in Processing (green is blob contour, red is blob bounding box)

- There could be functions from `Matrix` to `Matrix`. But we don't want a persistent matrix as output. We want something like `Matrix.Key` as output. That is an in-memory object that can be fed into the aural sonification.`Matrix.Key` does not carry shape information any longer (we could investigate if we can add that), which would be needed if we allowed inter-op with the graph elements, such as `Dim(...).size`.

- On the other hand, we could have two systems, a second persistent one for caching, with an actual `Matrix` object at the end. But this will cause further complications in the aural sonification, because we need to convert or cast systems.

- From the programmer's point of view, we should have a simple DSL to write the output matrix. This will, in any case, be a NetCDF file, so we can do away with the `Matrix` abstractions in that case.

- We have a system for "preparation", i.e. for off-line rendering: *FScape 2*. The question is, how feasible would it be to perform the blob analysis with the existing UGens (plus I/O)? If it costs effort but is feasible, we already have a functioning framework at hand (that could be refined with more UGens at a later stage).

Before making any decisions here, we should just write a small program to do the blob detection as we imagine it, to see if the effort is worth it and the approach works. We can probably run "global" block extraction, leading to the green contours in Fig. 17, and then "joining" vertical lines per blob, so that we obtain a one-dimensional signal (low/high margin plus centroid, possibly also horizontal "width" if we join horizontal lines as well; thus four or five scalar components over time).

Sketch of algorithm:

- Scan time/altitude segment (index latitude, index or average longitude).

- Apply blob analysis (globally); remove too small blobs etc.

- Reformulate blobs to be working as four/five time-based signals

- Decide on the number of "voices" needed. This could be one axis of the resulting matrix?
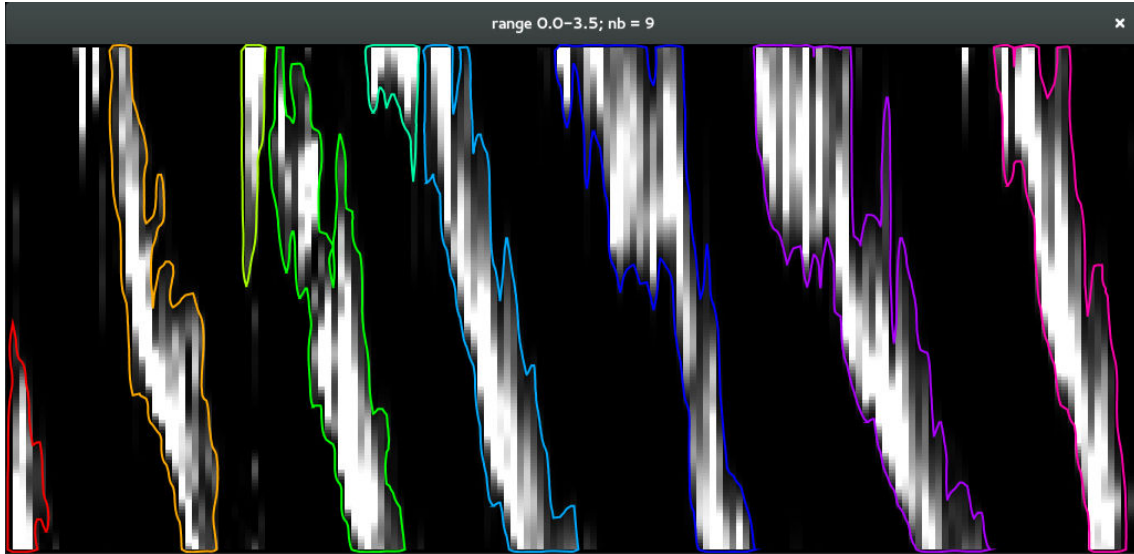
- Write out as matrix.

Figure 18: Taking a blob detection algorithm for Processing, and configuring it to work nicely with the anomalies data. Only hot anomalies are used here.

Look into the *Turbulence* code, because there was at some point a sort of blob detection in use: `https://github.com/iem-projects/sysson/blob/turbulence/src/main/scala/at/iem/sysson/turbulence/DataSets.scala#L162` – a blob here is a rectangle with latitude and longitude extent, so that doesn't help much.

After tuning the blob detection algorithm used in Fig. 17, a library for Processing written by Julien Gachadoat, one can get to a rather good separation, as shown in Fig. 18. There is still a slight problem with horizontal disconnection if months "fall out", which might be solved with a bit of horizontal blurring.

The next step is to turn these contours, which are currently polylines, into something that can be used in the sound synthesis process. One idea is to "scan" the blobs with horizontal and vertical "boxes" and then take the bounding boxes of the intersection shape, which should give us for every time step two boxes—a horizontal "extent" and a vertical "extent".