The SysSon Platform

Technical Report TR-2016-12-1
Institute of Electronic Music and Acoustics, Graz
(Status: completed)

Hanns Holger Rutz

December 2016

1 Presentation of QBO Sounds at WegC

The rendered examples from the November session at WegC have been uploaded to `https://soundcloud.com/syssonproject/sets/wegc_161110-neu`. The general feedback at the meeting was positive, and the next step will be to incorporate some of the issues and ideas raised, as well as to extend the sound models to the high-resolution data set.

Some decisions were agreed upon: representing cold temperatures with high frequencies or high register, and warm temperatures with low frequencies or low register; mapping altitudes to frequencies. With respect to the anomalies calculation, it was confirmed that one should normally use arithmetic means instead of the the median. Regarding the blob analysis, further tests are needed to explore more parameters of the analysis, since in the current sound model only centroid and vertical height had been used. In particular, it was suggested that the temporal duration or horizontal width would be more meaningful. Finally, while the cluster sonification (bank of sine oscillators) was well received, the kind of "echoes" or "reverberation" in the sound are confusing. A possible explanation for this phenomenon is that it constitutes an artefact from the fixed frequency relationship between the bands, and thus it could be alleviated by slightly randomising the centre frequencies or phases of the oscillators.

Second Feedback

At the beginning of December, we received a second written feedback by WegC staff based on the play-list created on the SoundCloud account (i.e., without giving further background information on the sound models): The blob based model using only sine oscillation was described as very clear in terms of perceiving the quasi-periodicity, the contour of the downward glissandi, and the overlap of glissandi. It was criticised that the "pitch" (possibly in combination with the non-existing timbre of the sine oscillation) was unpleasant, especially when having to listen to the sound repeatedly, rendering this timbre rather unsuitable for analysis purposes, while still remaining usable for demonstrating the QBO phenomenon.

In contrast, the timbre of the cluster model was perceived as more pleasant, while the sound is more confusing when subject to analysis, and small changes in the pitch are not easily audible. Also the overlap of the glissandi is more difficult to perceive. Again, the "echoes" in the sound are described as confusing.

The fixed rendering clearly reveals a disadvantage, as the click indicators for the months passing by were described as too low in volume, and for higher speed it was wished to remove the clicks altogether. This would be an indicator for implementing at least simple toggle buttons or sliders for the user parameters in the GUI. Likewise, being able to listen to the data at different tempos was notes as very useful.

Otherwise, the "figurative" model with two different timbres for hot/cold anomalies was described as having a better indication of the overlaps of the glissandi, whereas it is was more difficult to follow the precise contour in comparison to the sine oscillator example. Another suggestion given in the feedback was to allow the (interactive) isolation of "a sound level" (by which probably was meant the isolation of an altitude level and thus frequency band), in order to analyse the overlap of the glissandi and the periodicity of the QBO. For the next iteration, is was especially suggested to combine the temperature data with other variables such as precipitation or pressure, synchronised on the time line.

We have asked the person that gave this feedback to also address the different blob timbres (where the timbre changes from "flat" to more articulated based on the blob height). We have not received this additional feedback, yet.

2 Wrap up 2016

This sections tries to answer the question of the project report: “What has been achieved so far in the ongoing project year (milestones, intermediate steps; events)?”

- perform evaluation of a current situation with the platform; project several software milestones; implement milestone ’synth graph branching’ (implementation, tests, integration, documentation)
- implement milestones for improved sound model control (synth graph plotting, trace function)
- partial implementation ’automatic voice management’
- successively published four stable software releases (most recently 1.11.0)
- improvements and bug removals (5 tickets closed, 15 tickets opened in the main project; c. 30 tickets closed in related libraries); 76 commits to open source project
- documentation through an online help system for UGens as well as monthly technical reports regarding project process and sonification models
- first iteration QBO sonification models (quasi-biennial-oscillation); seven selected models discussed with WegC, feedback and planning of next iteration; sound examples on SoundCloud

3 Upcoming Work

This section tries to answer the question of the project report: “In the ongoing project year, what are the next steps planned (schedule, intermediate steps, milestones)?”

- at least two further iterations of sonification models (QBO: synchronous comparison with other variables; higher temporal resolution)
- translate models back into interactive user-controllable patch; installation at WegC as a tool usable in their regular praxis; dissemination as open source tool, including the developed sonification templates - documentation of the EEK translational processes; conference presentation (e.g. ICAD 2017 Pennsylvania; SMC 2017 Helsinki) - exchange with guest researchers in sonification at IEM (May 2017) - where feasible: implementation of further software milestones (matrix pre-processing DSL; interaction control / UI)

4 Matrix Pre-Processing DSL

The first use case of this proposal is to be able to integrate back into the main project the blob analysis that was run as an individual routine in the `https://github.com/iem-projects/sysson-experiments` project. That is, we want to define a number of steps from `val z = Var("blob")` to `val x = Var("anom"); val z = Blobs(z, ...)` that make it possible to express this transformation from within a sonification patch.

Relevant questions and issues:

- We should plug into the existing `buildAsyncInput` mechanism of `AuralProc`.

- We should, in turn, plug that into a mechanism that give UI feedback on asynchronous renderings taking place
- We should evaluate whether we can use *FScape-next* as a host architecture for specific matrix based rendering UGens.
- We need to find a way to cache the matrices rendered this way
- Related to the caching, we need to find a way to avoid having to recalculate the entire blob matrices when for example the time axis slice shrinks. I think this is already partly addressed by the current matrix-to-audio-file caching?

Analysis

The object `AnomaliesBlobs` in the experiments project was defined in terms of `NetCdfFileUtil.transformSelection`. This poses the problem of how to handle the dimensional meta-data inside the constraints of the rather flat data containers available in *FScape-next*.

The input matrix here has [time][lon][lat][alt] (perhaps in other order), the transform function is called with in-dims = [time, altitudes] and out-dims = [create(time), keep(longitudes), keep(latitudes), create(blob-data)]. Here is a first sketch of a pseudo-program:

```
val vIn    = Var("anom") // or 'MatrixIn'?
val dT     = Dim(vIn, "time")
val dAlt   = Dim(vIn, "altitudes")
// instead of massive multi-channel expansison, we 'linearise'
// the data here similar to the NetCdfFileUtil approach
val in     = vIn.playLinear(dT, dAlt) // or 'dAlt, dT'?
val thresh = (in - thresh).max(0)
val blobs  = Blobs(in = thresh, rows = dAlt.size, columns = dT.size,
   smooth = ???)
MatrixOut(key = "blobs", in = blobs, dims = ???)
```

Note that `playLinear` (or whatever it will be called) could also be interesting for the real-time synth-graph definition.

From this pseudo-program, the follow-up questions and issues are:

1. How do we wire this up with the real-time synth graph patch? We could define something like `MatrixIn(key = "blob")` Where at key 'blob' in the attribute map, we find the above FScape program.
2. Or, at key 'blob', we find instead a `Matrix[S]` and we extend the matrix type by one backed by an FScape program.
3. How do we now see the logical dimensions 'time' and 'altitudes' in the UI scanning process? We cannot scan an attribute map to find possibly used FScape programs that define matrices. We could require that these dimensions are redefined in the synth-graph object.
4. An alternative, theoretically, would be to embed the offline FScape processing code inside the main synth-graph function. But this will be messy with the change of imports inside the FScape function (among other things).
5. We should be able to refer to other patches in the same manner. For example, the above program could depend on another program that calculates the anomalies ad-hoc.

Item 2 could be independently realised at some point to create "lazy" transformed matrices. A counter-argument to this is that all "normal" expressions evaluate reasonably fast and synchronous, whereas this would introduce an asynchronous object which therefore is not available at all times. One could also imagine an `AudioCue.Obj[S]` that is backed by rendering through FScape, this would constitute a similar problem/challenge.

Further thoughts:

- We should keep other use cases in mind, for which methods on `Var` or `Dim` are not feasible or for which they would explode the number of pseudo UGens. For instance: Calculating the maximum and minimum of a matrix selection. This again raises the question of caching the value. Pseudo-program:

```
val vIn    = Var("anom") // or 'MatrixIn'?
val in     = vIn.playLinear()
val min    = RunningMin(in).last
val max    = RunningMax(in).last
ScalarOut(key = "min", in = min)
ScalarOut(key = "max", in = max)
```

- Now, since these programs can emit more than one thing, e.g. two different scalars in the previous example, or one or two matrices in the former example—it seems not smart to place an `FScape` object directly in the attribute map of the main sonification patch. How do we communicate the "output key"? We could of course specify two keys, like `MatrixIn(attr = "program", slot = "blobs")`. But it adds some confusion. If we think of the analogy of `ScanIn`, then we do not place a `Proc` directly in the sink's attribute map, but an `Output` instance instead.

- While `Output` instances aren't exactly pretty, they have an advantage where we cannot easily specify a secondary key; for example, when multiple output instances are grouped together in a `Folder`. Can we think of a similar scenario for FScape? There seem to be more differences than similarities:

  - We have already defined two entirely different types of data coming out of a patch: A matrix, and a scalar value. And there would most certainly also be the ordinary sound file.
  - There is no equivalent for `AuralObj`, i.e. a "black-box" with `play` functionality.

This all touches on the question of having higher-kinded types in *Sound Processes*, such as `Option[A]`, or `Future[A]`, or `Output[A]`. This is not a viable endeavour with the given resources now. We can of course always "erase" type information, so an FScape `Output` would be rather opaque. Or it carries the type-ID of the value produced?

Let's start over from the other side—the caching mechanism. The most precise definition would look at the sub-tree from the `ScalarOut` or `MatrixOut` object, and "flatten" all arguments on the way. For example, if we have a `MatrixIn` on the way, we would have to flatten that to a number of primitive values, looking up the key in the attribute map, perhaps finding a data-source, mapping that to a file name etc. In other words, a very complicated process. If we use the existing file-caching library, then this flat representation must also be compact and serialisable. The thing could really explore into the most generic caching mechanism, for example if one imagines that an FScape patch takes a timeline-bounce as input. Then what happens with synth-graph containing RNG based UGens?

A simpler approach is to put the burden of the correct caching key on the programmer, i.e. by requiring a `ScalarOut(key = "key", value = runMax.last, cache = matIn)`.

*

What would an `FScape.Output` look like?

```
trait Output[S <: Sys[S]] {
  def fscape: FScape[S]
  def key   : String
  def tpe   : Obj.Type
  // def tpeID: Int
  def value(implicit tx: S#Tx): Option[Obj[S]]
}
```

Then the optional value would appear and disappear as the program is started (UGens expanded) and completed. What if we run the program multiple times concurrently? There is no problem here because cache is transactional. We could add `Publisher` to `Output` if one wants to track the appearance and disappearance of the value.

The only sensitive place to calculate the cache key is from within the graph function. An easy solution could be to rely on the `requestAttribute` calls in UGen expansions. Additionally, we must track the request of RNGs, and if they happen, add the seed to the cache key. This simply algorithm can only produce false cache invalidations but no false cache verifications. Additionally, when the graph function variable is set, the cache must be cleared.

For matrices, we would add something similar to `requestAttribute`, and there we could add the information about removal of dimensional reductions etc. so that we construct indeed a valid cache key.