The SysSon Platform

Technical Report TR-2017-02-1
Institute of Electronic Music and Acoustics, Graz
(Status: in progress)

Hanns Holger Rutz

January 2017

1 Adding an Offline Preprocessing Stage

1.1 Next Steps

The remaining six steps will be covered in this technical report.

1.1.1 Accessing Sonification Instances

There is a persistent problem of understanding how to correctly model nested or extended objects. The `proc` inside a `Sonification` is one example. How to provide sonification context when creating an `AuralProc`? We have solved this "top-down" by enforcing the `AuralProc` to be created from inside an `AuralSonification`, essentially extending the implementation of the former type.

With `FScape` this problem reappears: If an aural-proc encounters a `Gen`, we will be able to construct an fscape-rendering from the gen-view-factory, but if the FScape graph itself wants to access the sonification, for example the `sources` map of matrices, how we do that? Unambiguous association is theoretically impossible, as there is no 1:1 relationship between an `FScape` instance and some outer `Sonification` (the same fscape could be part of two different sonifications, for example).

A simple work-around however is to hook into our custom gen-view-factory. This factory is globally installed to support SysSon UGens inside `FScape` by way of providing a `UGenGraphBuilderContextImpl`, somehow the equivalent of the aural-view. It is here that `requestInput` is invoked, and therefore this is the point where we can try to find the sonification. The relevant `requestInput` calls may then come from the real-time part of the aural-sonification, when resolving a `Gen`. In that part, we store the sonification in a transaction-local variable, and this can be recovered in `requestInput`.

1.2 Next Steps

In order to make good choices in the next steps, we should consider a specific example of offline processing, the pieces of which must be enabled in these steps to eventually write a fully functional program.

Say, we have a dimension reference and we average the matrix over that dimension and output it as a new matrix. For example, the input matrix has shape [time:180, lon:12, lat:36, alt:601], and we want to average across the longitudes, yielding shape [time:180, lat:36, alt:601]. So we have to average 180 times 12 windows of size $36 \times 601 = 21636$. That's an in-memory buffer of roughly 2 MB.

A hypothetical program is given in Fig. 1.

Here we introduce a hypothetical UGen that directly writes a NetCDF file. Eventually, we will want to provide an `FScape.Output` instead, probably naming the UGen `MkVar`.

TODO:

1.2.1 Meta Data UGens

TODO:

1.2.2 Dimensional Operators

TODO:

```
Graph {
  val v        = Var("var")
  val d        = Dim(v, "dim")
  val p        = v.playLinear()
  val isOk     = p >= 0 & p < 1000 // TODO, need isFillValue function
  val flt      = p * isOk
  val dSz      = d.size
  val tSz      = d.succSize  // good name?
  val cSz      = dSz * tSz
  val m        = Metro(cSz)
  val sum      = RunningWindowSum(flt , tSz, m)
  val count    = RunningWindowSum(isOk, tSz, m)
  val sumTrunc= ResizeWindow(sum  , size = cSz, start = cSz - tSz)
  val cntTrunc= ResizeWindow(count, size = cSz, start = cSz - tSz)
  val dataOut = sumTrunc / cntTrunc
  val specIn  = v.spec
  val specOut = specIn.drop(d)
  VarOut("file", specOut, in = dataOut)
}
```

Figure 1: Offline patch averaging over a dimension

1.2.3 Matrix Output

TODO:

1.2.4 Matrix Linkage from Offline to Real-Time

TODO:

1.2.5 Implementing Anomalies and Blobs