The SysSon Platform

Technical Report TR-2017-01-1
Institute of Electronic Music and Acoustics, Graz
(Status: completed)

Hanns Holger Rutz

January 2017

1 Adding an Offline Preprocessing Stage

The current workflow has been to experiment with matrix preprocessing directly in the IntelliJ IDE and outside of Mellite/SysSon. This is fine, because the IDE is much more powerful, and we have easier access to some of the utilities. However, the problem arises that we will need to make the results of these experiments available inside SysSon if we want to eventually allow the users to apply particular, newly developed preprocessing steps. We have done so with a separate menu-item to calculate anomalies, for example. We would now need to add another item for analysing the blobs. Obviously, this is not a scalable approach.

In the summer of 2016, I began to implement the next-generation FScape software, a toolkit for musical signal processing, using a UGen graph approach similar to ScalaCollider, but running offline. The back-end architecture uses the Akka-Stream framework, while the front-end API is very similar to ScalaCollider. This new FScape 2.x is already stable enough to use, as was demonstrated in various audio- and video-installations last year. Furthermore, it has a basic integration with SoundProcesses now. It is therefore an obvious choice to extend FScape with SysSon-specific UGens. A possible drawback is that the streams are weakly typed one dimensional signals (although channels can be bundled). There are a number of two-dimensional matrix and image processing UGens, but it requires that row size or width information must be explicitly passed into the respective UGens. Nevertheless, we estimate that it will be possible to implement the required UGens for SysSon matrices this way.

1.1 Implementation Steps

We have collected the required steps for a full implementation for processing matrices offline in SysSon below. As of February 2, the first five steps have been completed.

1. enhance SoundProcesses by "lazily" calculated objects
2. implement such a lazy object interface for FScape output
3. implement a caching mechanism for these objects
4. make it possible to use these cached instances as input to the real-time playback
5. implement a simple SysSon matrix reader in FScape
6. make `Sonification` instance matrices available in FScape
7. add commonly required meta data UGens, such as rank, size, dimensional information
8. add dimensional operators, such as transposition
9. add the possibility to *output* matrices from FScape
10. make it possible to use these cached matrices as input to the real-time sonification
11. add UGens necessary to complete existing processes (anomalies, blobs)

1.1.1 Lazily Calculated Objects

SoundProcesses now makes a simple provision to add support for lazily calculated objects (as would occur in *FScape*). A new `Obj` type `Gen` was added that acts as an opaque container for an eventually calculated peer object.

The only way to get hold of that value is to create a `GenView` instance. This instance, once created and until it is disposed, acts as an acquired lock on the caching/cached resource. A `GenView` is created by passing a `Gen` instance and an implicit `GenContext`. This context ensures that rendering processes are shared within a workspace between multiple occurrences of the `Gen` objects.

The `GenView` is similar to a transactional future. It has a method `value` that returns a `Option[Try[Obj[S]]`—corresponding to a `future.value`—a state that indicates the current progress or completion, and it is observable for state updates.

1.1.2 Lazily Objects in FScape

Similar to aural views, gen views are created by factories which are globally registered for a specific peer data type. In `FScape` we created a new object type `FScape.Output` that is a sub-type of `Gen`, and a default gen-view factory for this type. The view type is `OutputGenView` and it combines an instance of `FScape.Rendering` with an `FScape.Output`. We ensure that at maximum one rendering instance is running for an `FScape` object. All outputs, i.e. all lazily created by rendering the same graph function, are thus grouped together.

1.1.3 Caching Objects

When requesting an instance of `FScape.Rendering`, the graph function is converted into a cache key. The graph expansion is now explicitly divided into two stages, the first giving the collection of UGens, the second creating the stream nodes in the Akka framework. In order to calculate the cache key, we only have to execute the first stage. The structure captures the positions of the constants and UGens in the graph, and collects for each UGen an "auxiliary" object that represents the structural data uniquely identifying the environmental inputs of the UGen. For example, this may contain the path name, length and modification date of an input file, or a constant such as an integer or a string obtained during the first graph expansion stage.

Each entry in the cache is associated with a value structure that contains the peer data for all `FScape.Output` instances. When graph elements provide a coupling to an `FScape.Output`, they call `requestOutput` on the graph builder, providing the a `Output.Reader` that can *de-serialize* the peer data stored in a cache value. Correspondingly, `requestOutput` returns an `OutputRef`, eventually passed to the stream graph element, that has a method `complete` to be invoked once that stream graph element has determined the peer data. The value passed to `complete` is an instance of `Output.Writer`, capable of adding serialised data to the cache value when it is written.

So, when requesting an instance of `FScape.Rendering`, either a valid cache entry is found (through the structure key), and the "rendering" is actually not a running rendering but an encapsulation of the already known result, or if not, an actual rendering will begin, and when it terminates, the cache entry is written.

To summarise, `FScape.Rendering` denotes an ongoing or finished (read from cache) rendering process, it encapsulates the peer data of *all* connected outputs of an `FScape` object. The outputs are opaque objects, by way of `GenView(output)` one obtains `OutputGenView` instances for these, which in turn extract their particular peer data by querying the common `Rendering` instance. By calling `genView.dispose()`, access to that data is nominally lost, and the system may decide to wipe the cache data if needed (for example, if a given maximum cache size is exceeded).

1.1.4 Using Cached Objects in Real-Time Playback

Cached objects appear as values in another object's attribute map, so for example, the result of calculating a number might be a `Gen` whose peer data is an integer or a double, and this `Gen` instance will now appear in an attribute map instead of an "eager" `IntObj` or `DoubleObj`. This requires that especially the implementation of `AuralProc` recognises these new objects.

While a `UGenGraphBuilder.Value` has a `async` flag that could be useful here, we can first focus on input that must be resolved at graph expansion time, including scalar attributes without default value (and thus without a default number of channels). This is a simpler approach: When encountering a `Gen` during a `requestInput` call, we get hold of a `GenView`. We maintain a new map inside the aural-proc from attribute keys to gen-view instances. So, in `requestInput`, when this map does not contain the attribute key, we create the `GenView` and store it there. We then call `value` to see if the peer value is already available. If so, we look at this value and proceed as normal. If not, it means we need to wait for the rendering, and at this time, we cannot determine required properties such as the number-of-channels of a scalar input. We can throw a `MissingIn` exception, an established mechanism for deferring the playback of a proc until all required attribute inputs are present. Additionally, we need to track the completion of any `GenView` we have thus created. Similar to the `attrAdded` method which is invoked when an attribute is added to the proc's attribute map, we check if the completed view corresponds to one of the missing keys in the incomplete UGen builder state. If so, we attempt the build again.

A bit of complexity is added by the fact that we now must support `Gen` in a number of scenarios, from scalar values to in-memory buffers to streamed buffers. We have implemented all of these cases, but a future overhaul should look at a more general and DRY solution.

1.1.5 Simple SysSon Matrix Reader UGen

We have implemented new UGens for *FScape*, similar to the ones already available in *ScalaCollider*. `Var(key)` refers to a matrix found in the attribute map, and `playLinear()` produces a one-dimensional "flattened" stream of its contents:

```
val mat: DataSource.Variable[S] = ???
val f = FScape[S]
f.graph() = Graph {
  val v     = Var("var")
  val value = v.playLinear()
  val mx    = RunningMax(value).last
  val mn    = RunningMin(value).last
  MkDouble("max", mx)
  MkDouble("min", mn)
}
val outMx = f.outputs.add("max", DoubleObj)
val outMn = f.outputs.add("min", DoubleObj)
f.attr.put("var", mat)
```

In this example, we calculate the minimum and maximum element of a matrix and make it available through two outputs named `min` and `max`. These can be used in a subsequent real-time process as scalar inputs.

The implementation of the one-dimensional reader was quite challenging and is outlined in the next section.

1.2 Implementing a one-dimensional NetCDF reader

In order to stream arbitrary chunks of data from a NetCDF file into FScape, we need to overcome a limitation of the NetCDF API, namely that its own `read` command requires a regular, *rectangular* (hypercube) sectioning of the overall matrix, although of course, in the back-end inside the file, the data is stored in a flat manner. This problem hadn't occurred yet in the real-time part, because here we always require that one of the matrix dimensions is used as a temporal unrolling, while all other dimensions contribute to the the multi-channel-expanded signal. Implementing the 1D reader nevertheless has also benefit for the real-time part, as we will be able to add new UGens for example for a simple audification or "space-filling" usage of the matrix.

To solve this problem, we need an algorithm that converts, for a given (`off`, `len`) range in the flattened representation of a matrix (indices from zero to the number of elements in the matrix), this range into a sequence of regular hypercubes that can be then used to issue corresponding read commands and concatenate the individual results. For performance reasons, we want the number of commands to be as small as possible.

For example, we can calculate index vectors for the set of dimensions giving a linear index:

```
def calcIndices(off: Int, shape: Vector[Int]): Vector[Int] = {
  val modsDivs = shape zip shape.scanRight(1)(_ * _).tail
  modsDivs.map { case (mod, div) =>
    (off / div) % mod
  }
}
```

To develop a solution, let us say the shape is, for example, this, representing an array with rank 4 and 120 elements in total:

```
val sz  = Vector(2, 3, 4, 5)
val num = sz.product  // 120
```

A utility to print these index vectors for a range of linear offsets:

```
def printIndices(off: Int, len: Int): Unit =
  (off until (off + len)).map(calcIndices(_, sz))
    .map(_.mkString("[", ",␣", "]")).foreach(println)
```

We can generate all those vectors:

```
printIndices(0, num)
```

```
[0, 0, 0, 0]
[0, 0, 0, 1]
[0, 0, 0, 2]
[0, 0, 0, 3]
[0, 0, 0, 4]
[0, 0, 1, 0]
[0, 0, 1, 1]
[0, 0, 1, 2]
[0, 0, 1, 3]
[0, 0, 1, 4]
[0, 0, 2, 0]
[0, 0, 2, 1]
[0, 0, 2, 2]
[0, 0, 2, 3]
[0, 0, 2, 4]
[0, 0, 3, 0]
```

```
[0, 0, 3, 1]
[0, 0, 3, 2]
[0, 0, 3, 3]
[0, 0, 3, 4]
[0, 1, 0, 0]
...
[1, 2, 1, 4]
[1, 2, 2, 0]
[1, 2, 2, 1]
[1, 2, 2, 2]
[1, 2, 2, 3]
[1, 2, 2, 4]
[1, 2, 3, 0]
[1, 2, 3, 1]
[1, 2, 3, 2]
[1, 2, 3, 3]
[1, 2, 3, 4]
```

We can understand these indices as selecting sub-elements in the total section of the matrix we want to read. What we see in the preceding list is how the interleaving of indices works.

Let us look at an example chunk that should be read, the first six elements:

```
val off1 = 0
val len1 = 6
printIndices(off1, len1)
```

I will already partition the output by hand into the desired hypercubes:

```
// first hypercube or read
[0, 0, 0, 0]
[0, 0, 0, 1]
[0, 0, 0, 2]
[0, 0, 0, 3]
[0, 0, 0, 4]

// second hypercube or read
[0, 0, 1, 0]
```

So we need an algorithm that performs the analysis that gives us this partitioning into two read commands. The two commands are, reducing these sub-sequences of indices to *ranges*, `[0 to 0, 0 to 0, 0 to 0, 0 to 4]` (a hypercube of five elements), and `[0 to 0, 0 to 0, 1 to 0, 0 to 0]` (a hypercube of one element). It should also be obvious that these are not the same as a single `[0 to 0, 0 to 0, 1 to 0, 0 to 4]` would describe (wrongly) a hypercube of ten instead of six elements.

So the task is to define a method

```
def partition(shape: Vector[Int], off: Int, len: Int):
   List[Vector[Range]]
```

which outputs the correct list and uses the smallest possible list size. So for `off1` and `len1`, we have the expected result:

```
val res1 = List(
  Vector(0 to 0, 0 to 0, 0 to 0, 0 to 4),
  Vector(0 to 0, 0 to 0, 1 to 1, 0 to 0)
)
```

```
assert(res1.map(_.map(_.size).product).sum == len1)
```

A second example, elements at indices 6 until 22, with manual partitioning giving three hypercubes or read commands:

```
val off2 = 6
val len2 = 16
printIndices(off2, len2)

// first hypercube or read
[0, 0, 1, 1]
[0, 0, 1, 2]
[0, 0, 1, 3]
[0, 0, 1, 4]

// second hypercube or read
[0, 0, 2, 0]
[0, 0, 2, 1]
[0, 0, 2, 2]
[0, 0, 2, 3]
[0, 0, 2, 4]
[0, 0, 3, 0]
[0, 0, 3, 1]
[0, 0, 3, 2]
[0, 0, 3, 3]
[0, 0, 3, 4]

// third hypercube or read
[0, 1, 0, 0]
[0, 1, 0, 1]

expected result:

val res2 = List(
  Vector(0 to 0, 0 to 0, 1 to 1, 1 to 4),
  Vector(0 to 0, 0 to 0, 2 to 3, 0 to 4),
  Vector(0 to 0, 1 to 1, 0 to 0, 0 to 1)
)

assert(res2.map(_.map(_.size).product).sum == len2)
```

Solution

The idea of the solution algorithm is as follows:

- a **point-of-interest (poi)** is the left-most position at which two index representations differ (for example for `[0, 0, 0, 1]` and `[0, 1, 0, 0]` the poi is `1`)
- we recursively sub-divide the original (start, stop) linear index range
- we use motions in two directions, first by keeping the start constant and decreasing the stop through a special "ceil" operation on the start, later by keeping the stop constant and increasing the start through a special "floor" operation on the stop

- for each sub range, we calculate the poi of the boundaries, and we calculate "trunc" which is ceil or floor operation described above
- if this trunc value is identical to its input, we add the entire region and return
- otherwise we recurse
- the special "ceil" operation takes the previous start value and increases the element at the poi index and zeroes the subsequent elements; e.g. for `[0, 0, 1, 1]` and `poi = 2`, the ceil would be `[0, 0, 2, 0]`
- the special "floor" operation takes the previous stop value and zeroes the elements after the poi index; e.g. for `[0, 0, 1, 1]`, and `poi = 2`, the floor would be `[0, 0, 1, 0]`

Here is the basics of the implementation. First, a few utility functions:

```
def calcIndices(off: Int, shape: Vector[Int]): Vector[Int] = {
  val modsDivs = (shape, shape.scanRight(1)(_ * _).tail,
   shape.indices).zipped
  modsDivs.map { case (mod, div, idx) =>
    val x = off / div
    if (idx == 0) x else x % mod
  }
}

def calcPOI(a: Vector[Int], b: Vector[Int], min: Int): Int = {
  val res = (a.drop(min) zip b.drop(min)).indexWhere { case (ai,bi)
   => ai != bi }
  if (res < 0) a.size else res + min
}

def zipToRange(a: Vector[Int], b: Vector[Int]): Vector[Range] =
  (a, b).zipped.map { (ai, bi) =>
    require (ai <= bi)
    ai to bi
  }

def calcOff(a: Vector[Int], shape: Vector[Int]): Int = {
  val divs = shape.scanRight(1)(_ * _).tail
  (a, divs).zipped.map(_ * _).sum
}

def indexTrunc(a: Vector[Int], poi: Int, inc: Boolean): Vector[Int] =
  a.zipWithIndex.map { case (ai, i) =>
    if      (i < poi) ai
    else if (i > poi) 0
    else if (inc)     ai + 1
    else              ai
  }
```

Then the actual algorithm:

```
def partition(shape:Vector[Int],off:Int,len:Int):List[Vector[Range]]={
  val rankM = shape.size - 1

  def loop(start: Int, stop: Int, poiMin: Int, dir: Boolean,
           res0: List[Vector[Range]]): List[Vector[Range]] =
```

```
    if (start == stop) res0 else {
      val last = stop - 1
      val s0  = calcIndices(start, shape)
      val s1  = calcIndices(stop , shape)
      val s1m = calcIndices(last , shape)
      val poi = calcPOI(s0, s1m, poiMin)
      val ti  = if (dir) s0 else s1
      val to  = if (dir) s1 else s0
      val st  = if (poi >= rankM) to else indexTrunc(ti, poi, inc=dir)

      val trunc = calcOff(st, shape)
      val split = trunc != (if (dir) stop else start)

      if (split) {
        if (dir) {
          val res1= loop(start,trunc,poiMin=poi+1,dir=true ,res0=res0)
          loop          (trunc,stop ,poiMin=0    ,dir=false,res0=res1)
        } else {
          val s1tm = calcIndices(trunc - 1, shape)
          val res1 = zipToRange(s0, s1tm) :: res0
          loop          (trunc,stop ,poiMin=poi+1,dir=false,res0=res1)
        }
      } else {
        zipToRange(s0, s1m) :: res0
      }
    }

  loop(off, off + len, poiMin = 0, dir = true, res0 = Nil).reverse
}
```

Examples:

```
val sz  = Vector(2, 3, 4, 5)
partition(sz, 0, 6)

// result:
List(
  Vector(0 to 0, 0 to 0, 0 to 0, 0 to 4),  // first  hypercube
  Vector(0 to 0, 0 to 0, 1 to 1, 0 to 0)   // second hypercube
)

partition(sz, 6, 21)

// result:
List(
  Vector(0 to 0, 0 to 0, 1 to 1, 1 to 4),  // first  read
  Vector(0 to 0, 0 to 0, 2 to 3, 0 to 4),  // second read
  Vector(0 to 0, 1 to 1, 0 to 0, 0 to 4),  // third  read
  Vector(0 to 0, 1 to 1, 1 to 1, 0 to 1)   // fourth read
)
```

The maximum number of reads, if I am not mistaken, would be `2 * rank`. The (optimised) implementation is now integrated with the *LucreMatrix* library.

1.3 Next Steps

The remaining six steps will be covered in the next technical report.