Chapter 10

Laboratory Methods for Experimental Sonification

Till Bovermann, Julian Rohrhuber and Alberto de Campo

This chapter elaborates on sonification as an experimental method. It is based on the premise that there is no such thing as unconditional insight, no isolated discovery or invention; all research depends on methods. The understanding of their correct functioning depends on the context. Sonification as a relatively new ensemble of methods therefore requires the re-thinking and re-learning of commonly embraced understandings; a process that requires much experimentation.

Whoever has tried to understand something through sound knows that it opens up a maze full of both happy and unhappy surprises. For navigating this labyrinth, it is not sufficient to ask for the most effective tools to process data and output appropriate sounds through loudspeakers. Rather, sonification methods need to incrementally merge into the specific cultures of research, including learning, drafting, handling of complexity, and last but not least the communication within and between multiple communities. Sonification can be a great complement for creating multimodal approaches to interactive representation of data, models and processes, especially in contexts where phenomena are at stake that unfold in time, and where observation of parallel streams of events is desirable. The place where such a convergence may be found may be called a *sonification laboratory*, and this chapter discusses some aspects of its workings.

To begin with, what are the general requirements of such a working environment? A sonification laboratory must be flexible enough to allow for the development of new experimental methods for understanding phenomena through sound. It also must be a point of convergence between different methods, mindsets, and problem domains. Fortunately, today the core of such a laboratory is a computer, and in most cases its 'experimental equipment' is not hardware to be delivered by heavy duty vehicles, but is software which can be downloaded from online resources. This is convenient and flexible, but also a burden. It means that the division of labor between the development of tools, experiments, and theory cannot be taken for granted, and a given sonification toolset cannot be 'applied' without further knowledge;

within research, there is no such thing as ‘applied sonification’, as opposed to ‘theoretical sonification’. Participants in sonification projects need to acquire some familiarity with both the relevant discipline and the methods of auditory display. Only once a suitable solution is found and has settled into regular usage, these complications disappear into the background, like the medical display of a patient’s healthy pulse. Before this moment, both method and knowledge depend on each other like the proverbial chicken and egg. Because programming is an essential, but also sometimes intractable, part of developing sonifications, this chapter is dedicated to the software development aspect of sonification laboratory work. It begins with an indication of some common pitfalls and misconceptions. A number of sonification toolkits are discussed, together with music programming environments which can be useful for sonification research. The basics of programming are introduced with one such programming language, *SuperCollider*. Some basic sonification design issues are discussed in more detail, namely the relationship between time, order and sequence, and that between mapping and perception. Finally, four more complex cases of sonification designs are shown – vector spaces, trees, graphs, and algorithms – which may be helpful in the development process.

In order to allow both demonstration and discussion of complex and interesting cases, rather than comparing trivial examples between platforms, the examples are provided in a single computer language. In text-based languages, the program code also serves as precise readable documentation of the algorithms and the intentions behind them [17]. The examples given can therefore be implemented in other languages.

10.1 Programming as an interface between theory and laboratory practice

There is general agreement in the sonification community that the development of sonification methods requires the crossing of disciplinary boundaries. Just as the appropriate interpretation of visualized data requires training and theoretical background about the research questions under consideration, so does the interpretation of an auditory display. There are very few cases where sonification can just be applied as a standard tool without adaptation and understanding of its inner workings.

More knowledge, however, is required for productive work. This knowledge forms an intermediate stage, combining *know-how* and *know-why*. As laboratory studies have shown, the calibration and development of new means of display take up by far the most work in scientific research [24]. Both for arts and sciences, the conceptual re-thinking of methods and procedures is a constant activity. A *computer language* geared towards sound synthesis is a perfect medium for this kind of experimentation, as it can span the full scope from the development from first experiments to deeper investigations. It allows us to understand the non-trivial translations between data, theory, and perception, and permits a wider epistemic context (such as psychoacoustics, signal processing, and aesthetics) to be taken into account. Moreover, programming languages hold such knowledge in an operative form.

As algorithms are designed to specify processes, they dwell at the intersection between laboratory equipment and theory, as boundary objects that allow experimentation with different representation strategies. Some of what needs to be known in order to actively engage in the development and application of sonification methods is discussed in the subsequent sections in the form of generalized case studies.

10.1.1 Pitfalls and misconceptions

For clarification, this section discusses some common *pitfalls and misconceptions misconceptions* that tend to surface in a sonification laboratory environment. Each section title describes a misunderstanding, which is then disentangled in the section which follows:

»Data is an immediate given« Today, measured and digitized data appears as one of the rocks upon which science is built, both for its abundance and its apparent solidity. A working scientist will however tend to emphasize the challenge of finding appropriate data material, and will, wherever required, doubt its relevance. In sonification, one of the clearest indications of the tentative character of data is the amount of working hours that goes into reading the file formats in which the data is encoded, and finding appropriate representations for them, i.e., data structures that make the data accessible in meaningful ways. In order to do this, a working understanding of the domain is indispensable.

»Sonification can only be applied to data.« Often sonification is treated as if it were a method applied to data only. However, sonification is just as much relevant for the understanding of processes and their changing inner state, models of such processes, and algorithms in general. Sonification may help to perceptualize changes of states as well as unknowns and background assumptions. Using the terminology by the German historian of science Rheinberger [24], we can say that it is the distinction between technical things (those effects and facts which we know about and which form the methodological background of the investigation) and epistemic things (those things which are the partly unknown objects of investigation) that makes up the essence of any research. In the course of experimentation, as we clarify the initially fuzzy understanding of what the object of interest is exactly the notion of what does or does not belong to the object to be sonified can change dramatically. To merely "apply sonification to data" without taking into account what it represents would mean to assume this process to be completed already. Thus, many other sources than the common static numerical data can be interesting objects for sonification research.

»Sonification provides intuitive and direct access.« To understand something not yet known requires bringing the right aspects to attention: theoretical or formal reasoning, experimental work, informal conversation, and methods of display, such as diagrams, photographic traces, or sonification. It is very common to assume that acoustic or visual displays provide us somehow with more immediate or intuitive access to the object of research. This is a common pitfall: every sonification (just like an image) may be *read* in very different ways, requires acquaintance with both the represented domain and its representation conventions, and implies theoretical assumptions in all fields involved (i.e., the research domain, acoustics, sonification, interaction design, and computer science). This pitfall can be avoided by not taking acoustic insight for granted. The sonification laboratory needs to allow us to gradually learn to listen for specific aspects of the sound and to judge them in relation to their origin together with the sonification method. In such a process, intuition changes, and understanding of the data under exploration is gained indirectly.

»Data "time" and sonification time are the same.« Deciding which sound events of a sonification happen close together in time is the most fundamental design decision:

temporal proximity is the strongest cue for perceptual grouping (see section 10.4.1). By sticking to a seemingly compelling order (data time must be mapped to sonification time), one loses the heuristic flexibility of really experimenting with orderings which may seem more far-fetched, but may actually reveal unexpected phenomena. It can be helpful to make the difference between *sonification time* and *domain time* explicit; one way to do this formally is to use a sonification variable $\mathring{t}$ as opposed to t. For a discussion of sonification variables, see section 10.4.5.

»Sound design is secondary, mappings are arbitrary.« For details to emerge in sonifications, perceptual salience of the acoustic phenomena of interest is essential and depends critically on psychoacoustically well-informed design. Furthermore, perception is sensitive to domain specific meanings, so finding convincing metaphors can substantially increase accessibility. Stephen Barrass' *ear benders* [2] provide many interesting examples. Finally, "aesthetic intentions" can be a source of problems. If one assumes that listeners will prefer hearing traditional musical instruments over more abstract sounds, then pitch differences will likely sound "wrong" rather than interesting. If one then designs the sonifications to be more "music-like" (e.g., by quantizing pitches to the tempered scale and rhythms to a regular grid), one loses essential details, introduces potentially misleading artefacts, and will likely still not end up with something that is worthwhile music. It seems more advisable here to create opportunities for practicing more open-minded listening, which may be both epistemically and aesthetically rewarding once one begins to read the sonification's details fluently.

10.2 Overview of languages and systems

The history of sonification is also a history of laboratory practice. In fact, within the research community, a number of sonification systems have been implemented and described since the 1980s. They all differ in scope of features and limitations, as they were designed as laboratory equipment, intended for different specialized contexts. These software systems should be taken as integral part of the amalgam of experimental and thought processes, as "reified theories" (a term coined by Bachelard [1]), or rather as a complex mix between observables, documents, practices, and conventions [14, p. 18]. Some systems are now historic, meaning they run on operating systems that are now obsolete, while others are in current use, and thus alive and well; most of them are toolkits meant for integration into other (usually visualization) applications. Few are really open and easily extensible; some are specialized for very particular types of datasets.

The following sections look at dedicated toolkits for sonification, then focus on mature sound and music programming environments, as they have turned out to be very useful platforms for fluid experimentation with sonification design alternatives.

10.2.1 Dedicated toolkits for sonification

xSonify has been developed at NASA [7]; it is based on Java, and runs as a web service[1]. It aims at making space physics data more easily accessible to visually impaired people. Considering that it requires data to be in a special format, and that it only features rather simplistic sonification approaches (here called ‘modi’), it will likely only be used to play back NASA-prepared data and sonification designs.

The *Sonification Sandbox* [31] has intentionally limited range, but it covers that range well: Being written in Java, it is cross-platform; it generates MIDI output e.g., to be fed into any General MIDI synth (such as the internal synth on many sound cards). One can import data from CSV text files, and view these with visual graphs; a mapping editor lets users choose which data dimension to map to which sound parameter: Timbre (musical instruments), pitch (chromatic by default), amplitude, and (stereo) panning. One can select to hear an auditory reference grid (clicks) as context. It is very useful for learning basic concepts of parameter mapping sonification with simple data, and it may be sufficient for some auditory graph applications. Development is still continuing, as the release of version 6 (and later small updates) in 2010 shows.

Sandra Pauletto’s toolkit for Sonification [21] is based on PureData and has been used for several application domains: Electromyography data for Physiotherapy [22], helicopter flight data, and others. While it supports some data types well, adapting it for new data is slow, mainly because PureData is not a general-purpose programming language where reader classes for data files are easier to write.

SonifYer [27] is a standalone application for OSX, as well as a forum run by the sonification research group at Berne University of the Arts[2]. In development for several years now, it supports sonification of EEG, fMRI, and seismological data, all with elaborate user interfaces. As sound algorithms, it provides audification and FM-based parameter mapping; users can tweak the settings of these, apply EQ, and create recordings of the sonifications created for their data of interest.

SoniPy is a recent and quite ambitious project, written in the Python language [33]. Its initial development push in 2007 looked very promising, and it takes a very comprehensive approach at all the elements the authors consider necessary for a sonification programming environment. It is an open source project and is hosted at sourceforge[3], and may well evolve into a powerful and interesting sonification system.

All these toolkits and applications are limited in different ways, based on resources for development available to their creators, and the applications envisioned for them. They tend to do well what they were intended for, and allow users quick access to experimenting with existing sonification designs with little learning effort.

While learning music and sound programming environments will require more effort, especially from users with little experience in doing creative work with sound and programming, they already provide rich and efficient possibilities for sound synthesis, spatialization, real-time control, and user interaction. Such systems can become extremely versatile tools for the sonification laboratory context by adding what is necessary for access to the data and its

[1]http://spdf.gsfc.nasa.gov/research/sonification
[2]http://sonifyer.org/
[3]http://sourceforge.net/projects/sonipy

domain. To provide some more background, an overview of the three main families of music programming environments follows.

10.2.2 Music and sound programming environments

Computer Music researchers have been developing a rich variety of tools and languages for creating sound and music structures and processes since the 1950s. Current music and sound programming environments offer many features that are directly useful for sonification purposes as well. Mainly, three big families of programs have evolved, and most other music programming systems are conceptually similar to one of them.

Offline synthesis: MusicN to CSound

MusicN languages originated in 1957/58 from the Music I program developed at Bell Labs by Max Mathews and others. Music IV [18] already featured many central concepts in computer music languages such as the idea of a Unit Generator (UGen) as the building block for audio processes (unit generators can be, for example, oscillators, noises, filters, delay lines, or envelopes). As the first widely used incarnation, Music V was written in FORTRAN and was thus relatively easy to port to new computer architectures, from where it spawned a large number of descendants.

The main strand of successors in this family is *CSound*, developed at MIT Media Lab beginning in 1985 [29], which has been very popular in academic as well as dance computer music. Its main approach is to use very reduced language dialects for orchestra files (consisting of descriptions of DSP processes called instruments), and score files (descriptions of sequences of events that each call one specific instrument with specific parameters at specific times). A large number of programs were developed as compositional front-ends in order to write score files based on algorithmic procedures, such as Cecilia [23], Cmix, Common Lisp Music, and others. CSound created a complete ecosystem of surrounding software.

CSound has a very wide range of unit generators and thus synthesis possibilities, and a strong community; the CSound Book demonstrates its scope impressively [4]. However, for sonification, it has a few substantial disadvantages. Even though it is text-based, it uses specialized dialects for music, and thus is not a full-featured programming language. Any control logic and domain-specific logic would have to be built into other languages or applications, while CSound could provide a sound synthesis back-end. Being originally designed for offline rendering, and not built for high-performance real-time demands, it is not an ideal choice for real-time synthesis either. One should emphasize however that CSound is being maintained well and is available on very many platforms.

Graphical patching: Max/FTS to Max/MSP(/Jitter) to PD/GEM

The second big family of music software began with Miller Puckette's work at IRCAM on Max/FTS in the mid-1980s, which later evolved into Opcode Max, which eventually became Cycling'74's Max/MSP/Jitter environment[4]. In the mid-1990s, Puckette began developing

[4]http://cycling74.com/products/maxmspjitter/

an open source program called PureData (Pd), later extended with a graphics system called GEM.[5] All these programs share a metaphor of "patching cables", with essentially static object allocation of both DSP and control graphs. This approach was never intended to be a full programming language, but a simple facility to allow connecting multiple DSP processes written in lower-level (and thus more efficient) languages. With Max/FTS, for example, the programs actually ran on proprietary DSP cards. Thus, the usual procedure for making patches for more complex ideas often entails writing new Max or Pd objects in C. While these can run very efficiently if well written, special expertise is required, and the development process is rather slow, and takes the developer out of the Pd environment, thus reducing the simplicity and transparency of development.

In terms of sound synthesis, Max/MSP has a much more limited palette than CSound, though a range of user-written MSP objects exist. Support for graphics with Jitter has become very powerful, and there is a recent development of the integration of Max/MSP into the digital audio environment Ableton Live. Both Max and Pd have a strong (and partially overlapping) user base; the Pd base is somewhat smaller, having started later than Max. While Max is commercial software with professional support by a company, Pd is open-source software maintained by a large user community. Max runs on Mac OS X and Windows, but not on Linux, while Pd runs on Linux, Windows, and OS X.

Real-time text-based environments: SuperCollider, ChucK

The SuperCollider language today is a full-fledged interpreted computer language which was designed for precise real-time control of sound synthesis, spatialization, and interaction on many different levels. As much of this chapter uses this language, it is discussed in detail in section 10.3.

The ChucK language has been written by Ge Wang and Perry Cook, starting in 2002. It is still under development, exploring specific notions such as being strongly-timed. Like SuperCollider, it is intended mainly as a music-specific environment. While being cross-platform, and having interfacing options similar to SC3 and Max, it currently features a considerably smaller palette of unit generator choices. One advantage of ChucK is that it allows very fine-grained control over time; both synthesis and control can have single-sample precision.

10.3 SuperCollider: Building blocks for a sonification laboratory

10.3.1 Overview of SuperCollider

The SuperCollider language and real-time rendering system results from the idea of merging both real-time synthesis and musical structure generation into a single environment, using the same language. Like Max/PD, it can be said to be an indirect descendant of MusicN and CSound. From SuperCollider 1 (SC1) written by James McCartney in 1996 [19], it has gone through three complete rewriting cycles, thus the current version SC3 is a very

[5] http://puredata.info/

mature system. In version 2 (SC2) it inherited much of its language characteristics from the Smalltalk language; in SC3 [20] the language and the synthesis engine were split into a client/server architecture, and many features from other languages such as APL and Ruby were adopted as options.

As a modern and fully-fledged text-based programming language, SuperCollider is a flexible environment for many uses, including sonification. Sound synthesis is very efficient, and the range of unit generators available is quite wide. SC3 provides a GUI system with a variety of interface widgets. Its main emphasis, however, is on stable real-time synthesis. Having become open-source with version 3, it has since flourished. Today, it has quite active developer and user communities. SC3 currently runs on OS X and Linux. There is also a less complete port to Windows.

10.3.2 Program architecture

SuperCollider is divided into two processes: the language (*sclang*, also referred to as *client*) and the sound rendering engine (*scsynth*, also referred to as *server*). These two systems connect to each other via the networking protocol OpenSoundControl (*OSC*).[6]

SuperCollider is an interpreted fully-featured programming language. While its architecture is modeled on Smalltalk, its syntax is more like C++. Key features of the language include its ability to express and realize timing very accurately, its rapid prototyping capabilities, and the algorithmic building blocks for musical and other time-based compositions.

In contrast to sclang, the server, scsynth, is a program with a fixed architecture that was designed for highly efficient real-time sound-rendering purposes. Sound processes are created by means of synthesis graphs, which are built from a dynamically loaded library of unit generators (UGens); signals can be routed on audio and control buses, and soundfiles and other data can be kept in buffers.

This two-fold implementation has major benefits. First, other applications can use the sound server for rendering audio; Second, it scales well to multiple machines/processor cores, i.e., scsynth can run on one or more autonomous machines; and Third, decoupling sclang and scserver makes both very stable.

However, there are also some drawbacks to take into account. Firstly, there is always network latency involved, i.e., real-time control of synthesis parameters is delayed by the (sometimes solely virtual) network interface. Secondly, the network interface introduces an artificial bottleneck for information transfer, which in turn makes it hard to operate directly on a per sample basis. Thirdly, there is no direct access to server memory from sclang. (On OS X, this is possible by using the internal server, so one can choose one's compromises.)

SuperCollider can be extended easily by writing new classes in the SC language. There is a large collection of such extension libraries called Quarks, which can be updated and installed from within SC3.[7] One can also write new Unit Generators, although a large collection of these is already available as sc3-plugins.[8]

[6]http://opensoundcontrol.org/
[7]See the Quarks help file for details
[8]http://sourceforge.net/projects/SC3 plugins/

10.3.3 Coding styles

Thanks to the scope of its class library and its flexible syntax, SuperCollider offers many techniques to render and control sounds, and a variety of styles of expressing ideas in code. This short overview describes the basics of two styles (object style and pattern style), and shows differences in the way to introduce sound dynamics depending on external processes (i.e., data sonification). For a more detailed introduction to SuperCollider as a sound rendering and control language, please refer to the *SuperCollider Book* [32]. This also features a dedicated chapter on sonification with SuperCollider.

Object style Object-style sound control hides the network-based communication between client and server with an object-oriented approach. All rendering of sound takes place within the synthesis server (scsynth). The atom of sound synthesis is the unit generator (Ugen) which produces samples depending on its input parameters. UGens form the constituents of a fixed structure derived from a high-level description, the `SynthDef`, in sclang:

```
SynthDef(\pulse, { // create a synth definition named "pulse"
  |freq = 440, amp = 0.1| // controls that can be set at runtime
    Out.ar( // create an outlet for the sound
        0,      // on channel 0 (left)
        Pulse.ar( // play a pulsing signal
            freq // with the given frequency
        ) * amp // multiply it by the amp factor to determine its volume
    );
}).add; // add it to the pool of SynthDefs
```

In order to create a sound, we instantiate a `Synth` object parameterised by the SynthDef's name:

```
x = Synth(\pulse);
```

This does two things: firstly, it creates a synth object on the server which renders the sound described in the `pulse` synthesis definition, and secondly, it instantiates an object of type `Synth` on the client, a representation of the synth process on the server with which the language is able to control its parameters:

```
x.set(\freq, 936.236);  // set the frequency of the Synth
```

To stop the synthesis you can either evaluate

```
x.free;
```

or press the panic-button (hear sound example **S10.1**).[9] The latter will stop all synthesis processes, re-initialise the server, and stop all running tasks, whereas `x.free` properly releases only the synth process concerned and leaves everything else unaffected.

In this strategy, we can implement the simplest parameter mapping sonification possible in SuperCollider (see also section 10.4.2). Let's assume we have a dataset consisting of a one-dimensional array of numbers between 100 and 1000:

```
a = [ 191.73, 378.39, 649.01, 424.49, 883.94, 237.32, 677.15, 812.15 ];
```

[9] `<Cmd>-.` on OS X, `<Esc>` in gedit, `<Ctrl>-c <Ctrl>-s` in emacs, and `<alt>-.` on Windows.

With a construction called *Task*, a pauseable process that can run in parallel to the interactive shell, we are now able to step through this list and create a sound stream that changes its frequency according to the values in the list (hear sound example **S10.2**):

```
Task {
    // instantiate synth
    x = Synth(\pulse, [\freq, 20, \amp, 0]);
    0.1.wait;

    x.set(\amp, 0.1);         // turn up volume
    // step through the array
    a.do{|item| // go through each item in array a
        // set freq to current value
        x.set(\freq, item);

        // wait 0.1 seconds
        0.1.wait;
    };

    // remove synth
    x.free;
}.play;
```

The above SynthDef is continuous, i.e., it describes a sound that could continue forever. For many sound and sonification techniques, however, a sound with a pre-defined end is needed. This is done most simply with an envelope. It allows the generation of many very short sound events (sound grains). Such a grain can be defined as:

```
SynthDef(\sinegrain, {
  |out = 0, attack = 0.01, decay = 0.01, freq, pan = 0, amp = 0.5|

    var sound, env;

    // an amplitude envelope with fixed duration
    env = EnvGen.ar(Env.perc(attack, decay), doneAction: 2);

    // the underlying sound
    sound = FSinOsc.ar(freq);

    // use the envelope to control sound amplitude:
    sound = sound * (env * amp);

    // add stereo panning
    sound = Pan2.ar(sound, pan);

    // write to output bus
    Out.ar(out, sound)
}).add;
```

To render one such grain, we evaluate

```
Synth.grain(\sinegrain, [\freq, 4040, \pan, 1.0.rand2]);
```

Note that, in difference to the above example, the *grain* method creates an anonymous synth on the server, which cannot be modified while running. Thus, all its parameters are fixed when it is created. The grain is released automatically after the envelope is completed, i.e., the sound process stops and is removed from the server.

Using the dataset from above, a discrete parameter mapping sonification can be written like this (hear sound example **S10.3**):

```
Task {
    // step through the array
    a.do{|item|
        // create synth with freq parameter set to current value
        // and set decay parameter to slightly overlap with next grain
        Synth.grain(\sinegrain, [\freq, item, \attack, 0.001, \decay, 0.2]);

        0.1.wait; // wait 0.1 seconds between grain onsets
    };
}.play;
```

A third way to sonify a dataset is to first send it to a `Buffer` – a server-side storage for sequential data – and then use it as the source for dynamics control (hear sound example **S10.4**):

```
b = Buffer.loadCollection(
    server: s,
    collection: a,
    numChannels: 1,
    action: {"load completed".inform}
);

SynthDef(\bufferSon, {|out = 0, buf = 0, rate = 1, t_trig = 1, amp = 0.5|
    var value, synthesis;

    value = PlayBuf.ar(
        numChannels: 1,
        bufnum: buf,
        rate: rate/SampleRate.ir,
        trigger: t_trig,
        loop: 0
    );

    synthesis = Saw.ar(value);

    // write to outbus
    Out.ar(out, synthesis * amp);
}).add;

x = Synth(\bufferSon, [\buf, b])

x.set(\rate, 5000); // set rate in samples per second
x.set(\t_trig, 1);  // start from beginning
x.free;             // free the synthesis process
```

This style is relatively easy to adapt for audification by removing the synthesis process and writing the data directly to the audio output:

```

SynthDef(\bufferAud, {|out = 0, buf = 0, rate = 1, t_trig = 1, amp = 0.5|

    var synthesis = PlayBuf.ar(
        numChannels: 1,
        bufnum: buf,
        rate: rate/SampleRate.ir,
        trigger: t_trig,
        loop: 0
    );

    // write to output bus
    Out.ar(out, synthesis * amp)
}).add;
```

As the server's sample representation requires samples to be between -1.0 and 1.0, we have to make sure that the data is scaled accordingly. Also, a larger dataset is needed (see the chapter on audification, 12, for details). An artificially generated dataset might look like this:

```
a = {|i|cos(i**(sin(0.0175*i*i)))}!10000;
a.plot2; // show a graphical representation;
```

We can now load the dataset to the server and instantiate and control the synthesis process, just as we did in the example above (hear sound example **S10.5**):

```
b = Buffer.loadCollection(
    server: s,
    collection: a,
    numChannels: 1,
    action: {"load completed".inform}
);

// create synth
x = Synth(\bufferAud, [\buf, b, \rate, 44100]);

x.set(\t_trig, 1);              // restart
x.set(\rate, 200);              // adjust rate
x.set(\t_trig, 1, \rate, 400); // restart with adjusted rate
x.set(\t_trig, 1, \rate, 1500);

x.free;
```

Pattern style Patterns are a powerful option to generate and control sound synthesis processes in SuperCollider. A pattern is a high-level description of sequences of values that control a stream of sound events, which allows us to write, for example, a parameter mapping sonification in a way that also non-programmers can understand what is going on. Pattern-controlled synthesis is based on `Events`, defining a (predominately sonic) event with names and values for each parameter. Playing a single grain as defined in the object style paragraph then looks like this:

```
(instrument: \sinegrain, freq: 4040, pan: 1.0.rand2).play
```

When playing a pattern, it generates a sequence of events. The definition of the above discrete parameter mapping sonification in pattern style is (hear sound example **S10.6**):

```
a = [ 191.73, 378.39, 649.01, 424.49, 883.94, 237.32, 677.15, 812.15 ];
Pbind(
    \instrument, \sinegrain,
    \freq, Pseq( a ),  // a sequence  of the dataset a
    \attack, 0.001,    // and fixed values as desired
    \decay, 0.2,       // for the other parameters
    \dur, 0.1
).play
```

One benefit of the pattern style is that a wide range of these high-level controls already exist in the language. Let us assume the dataset under exploration is two-dimensional:

```
a = [
    [ 161.58, 395.14 ], [ 975.38, 918.96 ], [ 381.84, 293.27 ],
    [ 179.11, 146.75 ], [ 697.64, 439.80 ], [ 202.50, 571.75 ],
    [ 361.50, 985.79 ], [ 550.85, 767.34 ], [ 706.91, 901.56 ],
]
```

We can play the dataset by simply defining `a` with this dataset and evaluating the `Pbind` above. It results in two simultaneous streams of sound events, one for each pair (hear sound example **S10.7**). With a slight adjustment, we can even let the second data channel be played panned to the right (hear sound example **S10.8**):

```
Pbind(
    \instrument, \sinegrain,
    \freq, Pseq( a ),  // a sequence  of the data (a)
    \attack, 0.001,
    \decay, 0.2,
    \pan, [-1, 1],  // pan first channel to left output, second to right
    \dur, 0.1
).play
```

Comparison of styles

For modifying continuous sounds, and handling decisions unfolding in time very generally, ‘tasks’ are a very general and flexible tool. For creating streams from individual sounds, ‘patterns’ provide many options to express the implemented ideas in very concise terms. Depending on the context and personal preferences in thinking styles, one or other style may be better suited for the task at hand. The *Just In Time Programming Library* (JITLib) provides named proxies for tasks (Tdef), patterns (Pdef), and synths (Ndef), which allow to change running programs, simplify much technical administration, and thus can speed up development significantly.[10]

10.3.4 Interfacing

In this section, essential tools for loading data, recording the sonifications, and controlling the code from external processes are described. Due to the scope of this book, only the very essentials are covered. For a more in-depth overview on these themes, please consult the corresponding help pages, or the SuperCollider book [32].

Loading data Supposed, we have a dataset stored as comma-separated values (csv) in a text file called data.csv:

```
-0.49, 314.70,  964, 3.29
-0.27, 333.03,  979, 1.96
 0.11, 351.70, 1184, 5.18
-0.06, 117.13, 1261, 2.07
-0.02, 365.15,  897, 2.01
-0.03, 107.82, 1129, 2.24
-0.39, 342.26, 1232, 4.92
-0.29, 382.03,  993, 2.35
```

We can read these into SuperCollider with help of the CSVFileReader class:

```
a = CSVFileReader.readInterpret("data.csv");
a.postcs;  // post data
```

Each row of the dataset is now represented in SuperCollider as one array. These arrays are again collected in an enclosing array. A very simple sonification using the pattern method described in Section 10.3.3 looks like this:

```
// transpose the data representation
// now the inner arrays represent one row of the dataset
```

[10]For more information, see the JITLib help file, or the JITLib chapter in the SuperCollider book [32].

```
b = a.flop;

(
Pbind(
    \instrument, \sinegrain,
    \freq, Pseq([b[1], b[2]].flop, 2),
    \attack, 0.002,
    \decay, Pseq(b[3] * 0.1, inf),
    \pan, Pseq(b[0], inf),
    \dur, 0.1
).play
)
```

For very large data sets which are common in sonification it may be advisable to keep the data in a more efficiently readable format between sessions. For time series, such as EEG data, converting them to soundfiles will reduce load times considerably. For other cases, SuperCollider provides an archiving method for every object:

```
        // store data
a.writeArchive(path);
        // read data
a = Object.readArchive(path);
```

This can reduce load time by an order of two.

Recording sonifications SuperCollider provides easy and flexible ways to record real-time sonifications to soundfiles. Only the simplest case is covered here; please see the Server help file for more details.

```
        // start recording
s.record("/path/to/put/recording/test.wav");
        // run your sonification now ...
        // stop when done
s.stopRecording;
```

Control from external processes SuperCollider can be controlled from external applications by means of OpenSoundControl (OSC) [34]. Let us assume that an external program sends OSC messages in the following format to SC3[11]:

```
/data, iff 42 23.0 3.1415
```

You can set up a listener for this message with:

```
OSCresponder(nil, "/data", {|time, responder, message|
        "message % arrived at %\n".postf(message, time);
}).add;
```

We leave it as an exercise to the reader to integrate this into a sonification process. In-depth discussions of many sonification designs and their implementations in SC3 can be found in Bovermann [5] and de Campo [9].

[11]Note that SuperCollider's default port for incoming OSC messages is 57120.

10.4 Example laboratory workflows and guidelines for working on sonification designs

This section discusses many of the common concerns in creating, exploring and experimenting with sonification designs and how to integrate them in a laboratory workflow. Here, theoretical considerations alternate with examples that are generic enough to make it easy to adapt them to different contexts.

What is usually interesting about specific data sets is discovering the possible relationships between their constituents; some of these relations may be already established, whereas others may not yet be evident. Perceptualization is the systematic attempt to represent such relationships in data (or generally, objects under study) such that relationships between the constituents of the sensory rendering emerge in perception. This means that an observer notices *gestalts*, which may confirm or disprove hypotheses about relationships in the data. This process relies on human perceptual and cognitive abilities; most importantly that of organizing sensory events into larger groups. In auditory perception, this grouping of individual events depends on their perceptual parameters and their relationships, i.e., mainly inter-similarities and proximities.

In a successful sonification design, the relationships within the local dynamic sound structure (the proximal cues) allow a listener to infer insights into the data being sonified, effectively creating what can be considered distal cues. As there are very many possible variants of sonification design, finding those that can best be tuned to be very sensitive to the relationships of interest, however, is a nontrivial methodological problem.

The *Sonification Design Space Map (SDSM)* [8, 9] aims to help in the process of developing sonification designs. Put very briefly, while the working hypotheses evolve, as the sonification designs become more and more sophisticated, one repeatedly answers three questions:

1. How many data points are likely necessary for patterns to emerge perceptually?
2. How many and which data properties should be represented in the design?
3. How many parallel sound-generating streams should the design consist of?

Based on the answers, the SDSM recommends making sure the desired number of data points is rendered within a time window of 3–10 seconds (in order to fit within non-categorical echoic memory) [28], and it recommends suitable strategies (from Continuous, Discrete-Point, and Model-based approaches). As Figure 10.1 shows, changes in the answers correspond to movements of the current working location on the map: Zooming in to fewer data points for more detail moves it to the left, zooming out moves it to the right; displaying more data dimensions moves it up, while using more or fewer parallel sound streams moves it in the z-axis.

In practice, time spent exploring design alternatives is well spent, and helps by clarifying which (seemingly natural) implicit decisions are being taken as a design evolves. The process of exchange and discussion in a hypothetical research team, letting clearer questions evolve as the sonification designs become more and more sensitive to latent structures in the data, process or model under study, is of fundamental importance. It can be considered the equivalent of the common experience in laboratory work that much of the total work time is absorbed by setting up and calibrating equipment, until the experimental setup is fully

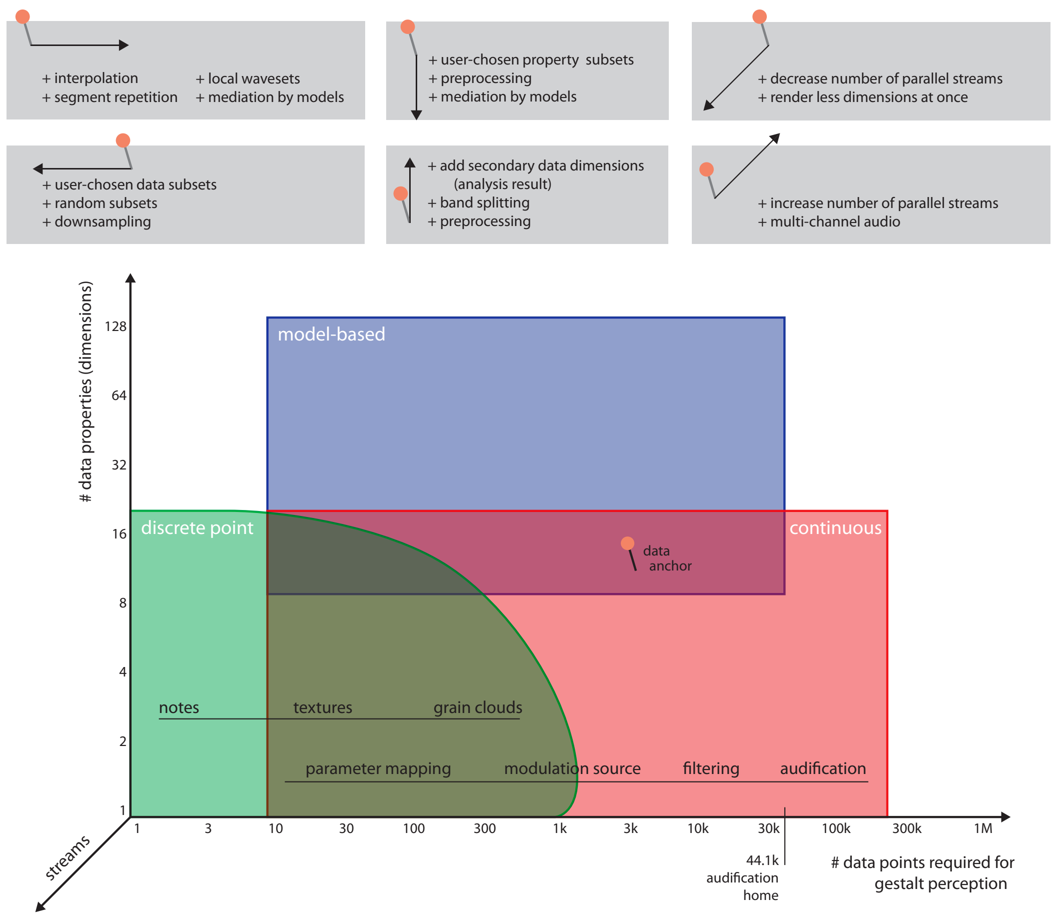

Figure 10.1: The Sonification Design Space Map.

"tuned", while by comparison, much less time is usually spent with actual measurement runs themselves. Such calibration processes may involve generating appropriate test data, as well as doing listening tests and training.

There now follow three sections explaining typical scenarios, in which data sonification workers may find themselves. As proximity in time is the property that creates the strongest perceptual grouping, especially in sound, the first section covers data ordering concepts and the handling of time in sonification. The second section discusses fundamental issues of mapping of data dimensions to sound properties via synthesis parameters, which requires taking perceptual principles into account. The later three sections address more complex cases, which raise more complex sets of questions.

10.4.1 Basics 1: Order, sequence, and time

In any data under study, we always need to decide which relations can be ordered and according to what criteria. Data from different geographic locations, for instance, may be ordered by longitude, latitude and/or altitude. Mapping a data order to a rendering order (such as altitude to a time sequence) means treating one dimension differently from the others.

As temporal order is the strongest cue for grouping events perceptually in the sonic domain, experimenting with mappings of different possible data orders to the temporal order of sounds can be very fruitful.

Example solutions

Here is a very simple example to demonstrate this constellation. Assume for simplicity that the domain data points are all single values, and they come in a two dimensional order, represented by an array:

```
a = [
    [ 0.97, 0.05, -0.22, 0.19, 0.53, -0.21, 0.54, 0.1, -0.35, 0.04 ],
    [ -0.07, 0.19,  0.67, 0.05, -0.91, 0.1,  -0.8, -0.21, 1, -0.17 ],
    [ 0.67, -0.05, -0.07, -0.05, 0.97, -0.65, -0.21, -0.8, 0.79, 0.75 ]
];
```

Two ordered dimensions are obvious, horizontal index, and vertical index, and a third one is implied: the magnitude of the individual numbers at each index pair. Depending on where this data came from, the dimensions may correlate with each other, and others may be implied, some of which may be unknown.

For experimentation, we define a very simple synthesis structure that creates percussive decaying sound events. This is just sufficient for mapping data values to the most sensitive perceptual property of sound - pitch - and experimenting with different ordering strategies.

```
SynthDef(\x, { |freq = 440, amp = 0.1, sustain = 1.0, out = 0|
    var sound = SinOsc.ar(freq);
    var env = EnvGen.kr(Env.perc(0.01, sustain, amp), doneAction: 2);
    Out.ar(out, sound * env);
}).add;
```

This sound event has four parameters: amplitude, sustain (duration of sound), frequency, and output channel number (assuming one uses a multichannel audio system).

As there is no inherent preferred ordering in the data, a beginning strategy would be to experiment with a number of possible orderings to develop a sense of familiarity with the data and its possibilities, and noting any interesting details that may emerge.

Relating time sequence with horizontal index, and frequency to the data value at that point, we can begin by playing only the first line of the data set (hear sound example **S10.9**):

```
// define a mapping from number value to frequency:
f = { |x| x.linexp(-1, 1, 250, 1000) };
Task {
    var line = a[0]; // first line of data
    line.do { |val|
        (instrument: \x, freq: f.value(val)).play;
        0.3.wait;
    }
}.play;
```

Next, we play all three lines, with a short pause between events and a longer pause between lines, to maintain the second order (hear sound example **S10.10**):

```
Task {
    a.do { |line|
        line.do { |val|
            (instrument: \x, freq: f.value(val)).play;
            0.1.wait;
    },
    0.3.wait;
}.play;
```

When we sort each line before playing it, the order in each line is replaced with order by magnitude (hear sound example **S10.11**):

```
Task {
    a.do { |line|
        line.copy.sort.do { |val|
            (instrument: \x, freq: f.value(val)).play;
            0.1.wait;
    };
    0.3.wait;
    }
}.play;
```

We play each line as one chord, so the order within each line becomes irrelevant (hear sound example **S10.12**):

```
Task {
    a.do { |line|
        line.do { |val|
            (instrument: \x, freq: f.value(val)).play; // no wait time here
    };
    0.3.wait;
}.play;
```

We can also use vertical order, and play a sequence of all columns (hear sound example **S10.13**):

```
Task {
    var cols = a.flop; // swap rows <-> columns
    cols.do { |col|
        col.do { |val|
            (instrument: \x, freq: f.value(val)).play;
            0.1.wait;     // comment out for 3-note chords
        };
        0.3.wait;
    };
}.play;
```

Finally, we play all values in ascending order (hear sound example **S10.14**):

```
Task {
    var all = a.flat.sort;
    all.do { |val|
        (instrument: \x, freq: f.value(val)).play;
        0.1.wait;
    };
}.play;
```

All these variants bring different aspects to the foreground: Hearing each line as a melody allows the listener to compare the overall shapes of the three lines. Hearing each column as a three note arpeggio permits comparing columns for similarities. Hearing each column as a chord brings similarity of the (unordered) sets of elements in each column into focus. Hearing each line sorted enables observation of what value ranges in each line values are denser or sparser. Sorting the entire set of values applies that observation to the entire dataset.

Discussion

It is productive to be aware of explicit orderable and un-orderable dimensions. Simple experiments help the designer to learn, to adjust, and to develop how these dimensions interrelate. Writing systematic variants of one experiment brings to the surface nuances that may become central evidence once discovered. For instance, with every new ordering, different structures might emerge. With unknown data, cultivating awareness of alternative orderings and data structures will help for fruitful experimentation and for learning to distinguish the impact of differences on a given sonification. Note that there are many psychoacoustic peculiarities in timing – for instance, parallel streams of sound may emerge or not dependent on tempo, and a series of events may fuse into a continuum.

10.4.2 Basics 2: Mapping and perception

Every sonification design involves decisions regarding how the subject of study determines audible aspects of the perceptible representation. It is thereby necessary to take into account the psychoacoustic and perceptual concepts underlying sound design decisions. Here, the discussion of these facts is very brief; for a more in-depth view see chapter 3, for a longer discussion of auditory dimensions see chapter 4, finally, for an introduction to mapping and scaling, see chapter 2.

Audible aspects of rendered sound may serve a number of different purposes:

1. Analogic display - a data dimension is mapped to a synthesis parameter which is easy to recognise and follow perceptually. Pitch is the most common choice here; timbral variation by modulation techniques is also well suited for creating rich, non-categorical variety.
2. Labelling a stream – this is needed for distinguishing categories, especially when several parallel streams are used. Many designers use instrumental timbres here; we find that spatial position is well suited as well, especially when using multiple loudspeakers as distinct physical sound sources.
3. Context information/orientation – this is the mapping non-data into the rendering, such as using clicks to represent a time grid, or creating pitch grids for reference.

Tuning the ranges of auditory display parameters plays a central role in parameter mapping sonification (see chapter 15), but indirectly it plays into all other approaches as well. Physical parameters, such as frequency and amplitude of a vibration, are often spoken of in identical terms to synthesis processes, as in the frequency and amplitude of an oscillator. They typically correspond to perceived sound properties, like pitch and loudness, but the correspondence is not always a simple one. First, we tend to perceive amounts of change relative to the absolute value; a change of 6% of frequency will sound like a tempered half-step in most of the audible frequency range. Second, small differences can be inaudible; the limit where half the test subjects say a pair of tones is the same and the other half says they are different is called the *just noticeable difference* (JND). The literature generally gives pitch JND as approximately

0.2 half-steps or about 1% frequency difference (degrading towards very high and very low pitches, and for very soft tones) and loudness differences of around 1 dB (again, worse for very soft sounds, and potentially finer for loud sounds). However, this will vary depending on the context: when designing sonifications, one can always create test data to learn which data differences will be audible with the current design. In the example in section 10.4.1, the numerical value of each data point was mapped to an exponential frequency range of 250 – 1000 Hz. In the first data row, the smallest difference between two values is 0.01. We may ask now whether this is audible with the given design (hear sound examples **S10.15** and **S10.16**):

```
// alternate the two close values
Task { loop {
    [0.53, 0.54].do { |val|
        (instrument: \x, freq: f.value(val)).play;
        0.1.wait;
    }
} }.play;

        // then switch between different mappings:
f = { |x| x.linexp(-1, 1, 250, 1000) }; // mapping as it was
f = { |x| x.linexp(-1, 1, 500, 1000) };  // narrower
f = { |x| x.linexp(-1, 1, 50, 10000) }; // much wider

        // run entire dataset with new mapping:
Task {
    a.do { |line|
        line.do { |val|
            (instrument: \x, freq: f.value(val)).play;
            0.1.wait;
        }
    };
    0.3.wait;
}.play;
```

When playing the entire dataset with the new wider mapping, a new problem emerges: the higher sounds appear louder than the lower ones. The human ear's perception of loudness of sine tones depends on their frequencies. This nonlinear sensitivity is measured experimentally in the equal loudness contours (see also chapter 3). In SC3, the UGen `AmpComp` models this: based on the frequency value, it generates boost or attenuation factors to balance the sound's loudness. The following SynthDef exemplifies its usage:

```
SynthDef(\x, { |freq = 440, amp = 0.1, sustain = 1.0, out = 0|
    var sound = SinOsc.ar(freq);
    var ampcomp = AmpComp.kr(freq.max(50));  // compensation factor
    var env = EnvGen.kr(Env.perc(0.01, sustain, amp), doneAction: 2);
    Out.ar(out, sound * ampcomp * env);
}).add;
```

So far also it is assumed that the sound events' duration (its sustain) is constant (at 1 second). This was just an arbitrary starting point; when one wants to render more sound events into the same time periods, shorter sounds have less overlap and thus produce a clearer sound shape. However, one loses resolution of pitch, because pitch perception becomes more vague with shorter sounds (hear sound example **S10.17**).

```
Task {
    a.do { |line|
        line.do { |val|
            (instrument: \x, freq: f.value(val), sustain: 0.3).play;
            0.1.wait;
        };
    };
    0.3.wait;
}.play;
```

We can also decide to assume that each of the three lines can have a different meaning; then we could map, for example, the values in the first line to frequency, the second to sustain, and the third to amplitude (hear sound example **S10.18**):

```
Task {
    var cols = a.flop; // swap rows <-> columns
    cols.do { |vals|
        var freq = vals[0].linexp(-1, 1, 300, 1000);
        var sustain = vals[1].linexp(-1, 1, 0.1, 1.0);
        var amp = vals[2].linexp(-1, 1, 0.03, 0.3);

        (instrument: \x,
            freq: freq,
            sustain: sustain,
            amp: amp
        ).play;
        0.2.wait;
    };
}.play;
```

Finally, we make a different set of assumptions, which leads to different meanings and mappings again: If we interpret the second line to be a comparable parameter to the first, and the third line to represent how important the contribution of the second line is, we can map the three lines to basic frequency, modulation frequency, and modulation depth (hear sound example **S10.19**):

```
SynthDef(\xmod, { |freq = 440, modfreq = 440, moddepth = 0,
    amp = 0.1, sustain = 1.0, out = 0|
    var mod = SinOsc.ar(modfreq) * moddepth;
    var sound = SinOsc.ar(freq, mod);
    var env = EnvGen.kr(Env.perc(0.01, sustain, amp), doneAction: 2);
    Out.ar(out, sound * env);
}).add;

Task {
    var cols = a.flop; // swap rows <-> columns
    cols.do { |vals|
        var freq = vals[0].linexp(-1, 1, 250, 1000);
        var modfreq = vals[1].linexp(-1, 1, 250, 1000);
        var moddepth = vals[2].linexp(-1, 1, 0.1, 4);
        (instrument: \xmod,
            modfreq: modfreq,
            moddepth: moddepth,
            freq: freq,
            sustain: 0.3,
            amp: 0.1
        ).postln.play;
        0.2.wait;
    };
}.play;
```

Discussion

Tuning display processes in such a way that they are easy to read perceptually is by no means trivial. Many synthesis and spatialization parameters behave in subtly or drastically different ways and lend themselves to different purposes in mappings. For example, while recurrent patterns (even if shifted and scaled) are relatively easy to discern when mapped to pitch, mapping them to loudness would make recognizing them more difficult, whereas mapping them to spatial positions would reduce the chances of false assignment.

In the sonification laboratory, it is good practice to test the audible representation, in much the same way as one would other methods. By creating or selecting well-understood test datasets, and verifying that the intended audience can confidently hear the expected level of perceptual detail, one can verify its basic viability for the context under study.

10.4.3 Vector spaces

While a given dataset is never entirely without semantics, it can be heuristically useful to abstract from what is known in order to identify new relations and thus build up a new semantic layer. One may quite often be confronted with the task of sonifying a numerical dataset that is embedded into a high-dimensional vector space, where axis descriptions were actively pruned. In other words, the orientation of the vector basis is arbitrary, thus carrying no particular meaning. We would like to be able to experiment with different approaches that take this arbitrariness into account.

Example solutions

Simple mapping approaches may consist of random mapping choices, picking one dimension for time ordering, and choosing others for mapping to control parameters of the sound display. For examples on these, see sections 10.4.1 and 10.4.2.

Principal Component Analysis (PCA) A related linearization method for datasets embedded into vector spaces is to find a linear transformation:

$$\hat{x}_j = \sum_i (x_j - \hat{o}) \cdot a_i$$

(with x_j data item, $\hat{x}_j$ transformed data item, $\hat{o}$ new origin, and a_i the i-th basis vector) that – based on either domain-specific knowledge or based on the actual dataset – makes sense. One option for deriving dataset-inherent knowledge is Principal Component Analysis (PCA) which returns basis vectors ordered by variances in their direction.[12] For more details on this approach, see chapter 8. The `pc1` method on `SequenceableCollection`, provided with the *MathLib* quark, calculates an estimation of the first principal component for a given, previously whitened dataset:

```
// a 2-d dataset with two quasi-gaussian distributions
d = {#[[-1, -0.5], [1, 0.5]].choose + ({0.95.gauss}!2)}!10000;
p = d.pc1; // first principal component
```

To estimate the next principal component, we have to subtract the fraction of the first one and do the estimation again:

```
f = d.collect{|x|
    var proj;
```

[12] While nearly any dataset could be interpreted as a vector-space, PCA is more useful for high-dimensional data.

```
    proj = ((v * x).sum * v); // projection of x to v
    x-proj; // remove projection from data item
}
p = d.pc1; // second principal component
```

For computing all principal components of a multidimensional dataset, we can apply the process recursively:

```
// compute components for n-dimensional datasets
q = ();

q.data; // the data
q.dim = q.data.shape.last
q.subtractPC = {|q, data, pc|
    var proj;
    data.collect{|x|
        proj = ((pc * x).sum * pc); // projection of x to v
        x-proj; // remove projection from data item
    };
}

// recursive function to calculate the steps needed
q.computePCs = {|q, data, pcs, dims|
    var pc;

    (dims > 1).if({
        pc = data.pc1;
        pcs[data.shape.last-dims] = pc;
        pcs = q.computePCs(q.subtractPC(data, pc), pcs, dims-1);
    }, {
        pc = data.pc1;
        pcs[data.shape.last-dims] = pc;
    });
    pcs;
}

// calculate and benchmark. This might take a while
{q.pcs = q.computePCs(q.data, 0!q.dim!q.dim, q.dim)}.bench;
```

This dimensional reduction, respectively dimension reorganization process alters the dataset's representation but not its (semantic) content. After this kind of pre-processing, strategies as introduced in Sections 10.4.1 and Section 10.4.2 become applicable again.

For more complex, statistical analysis methods, we suggest using tools like octave[13] or NumPy/SciPy[14] as they already implement well-tested methods for this, which otherwise have to be implemented and tested in SuperCollider.

Distance If the absolute data values are not of interest (e.g., because of the absence of a reference point), relative information might be worth sonifying. One such information is the distance between data items. It implicitly contains information like dataset density (both global and local), variance and outliers. The distance matrix for all data items can be computed by:

```
q = ();

// the dataset
q.data = {|dim = 4|
    ({{1.0.rand}!dim + 5}!100) ++ ({{10.0.rand}!dim}!100)
}.value
```

[13]`http://www.gnu.org/software/octave/`
[14]`www.scipy.org/`

```


// function to compute the distance between two points
q.dist = {|q, a, b|
    (a-b).squared.sum.sqrt;
}

// compute the distance matrix for the dataset
q.distanceMatrix = {|data|
    var size = data.size;
    var outMatrix = 0!size!size; // fill a matrix with zeroes
    var dist;

    data.do{|item, i|
        i.do{|j|
            dist = q.dist(item, data[j]);
            outMatrix[i][j] = dist;
            outMatrix[j][i] = dist;
        }
    };

    outMatrix
}.value(q.data)
```

Since the resulting matrix can be recognised as an (undirected) graph, it can be sonified for example by methods as described in the upcoming section 10.4.4.

Model-based sonification A third set of methods to sonify vector spaces are model-based sonifications as introduced in chapter 16. Next is shown an example for a data sonogram [15], which uses the data and distance algorithm described in the previous paragraph.

```
SynthDef(\ping, {|freq = 2000, amp=1|
    var src = Pulse.ar(freq);
    var env = EnvGen.kr(Env.perc(0.0001, 0.01), 1, doneAction: 2) * amp;
    Out.ar(0, (src * env) ! 2)
}).add;

(
q = q ? ();
// generate score of OSC messages, sort and play
q.createScore = {|q, dataset, rSpeed=1, impactPos|
    var dist, onset, amp;

    // set impactPos to a useful value - best by user interaction
    impactPos = impactPos ? 0.0.dup(dataset.shape.last);


        // for each data item, compute its distance from the impact center and
        // create an event according to it in the score*/
        // first ping represents impact
    [[0, [\s_new, \ping, -1, 0, 0, \freq, 1500]]]
        ++ dataset.collect{|row|
                // set onset time proportional to the distance
            dist = q.dist(row, impactPos);
            onset = dist * rSpeed;

                // compute amplitude according to distance from impactPos
                // less excitation > less amplitude
            amp = dist.squared.reciprocal;

                // finally, create the event
            [onset, [\s_new, \ping, -1, 0, 0, \amp, amp]];
        };
};
)
```

```

    // use the above defined function with a fixed position
q.scoreData = q.createScore(q.data, 2, 2!4);

    // generate a score, sort, and play it
Score(q.scoreData).sort.play
)
```

In keeping with the scope of this chapter, this is quite a simplistic implementation. However, it can be easily extended to feature also additional values of a dataset, e.g., by mapping them to the frequency of the individual sonic grain. Also, it would be worth implementing a GUI representation, allowing the user to literally tap the dataset at various positions.

Discussion

While semantics are an inherent part of any dataset, it is sometimes beneficial to consciously neglect it. In the analysis of the insights gained, however, the semantics should be again considered in order to understand what the new structural findings could really mean. While keeping track of relevant details, methods like the ones discussed above permit a process of gradually shifting between structure and meaning and of moving between different domains.

10.4.4 Trees and graphs: towards sonifying higher order structures

Trees and graphs in general are characterised by the fact that they provide more than one way to access or traverse them. As we can typically reach a node from more than one other node, there is an inherent choice to be made. This also demonstrates that data is contextual and access not immediate and univocal. Traversing a graph can be a non-trivial task – as exemplified by Euler's famous problem from 1735, of how to cross all *Seven Bridges of Koenigsberg* only once, is a good example. Any grammatical structure, such as a computer program, or even this very sentence, implies graphs.

	0	1	2	3	4
0	0	0	0	1	0
1	1	0	1	0	1
2	0	0	1	0	0
3	0	0	1	0	1
4	0	0	1	1	0

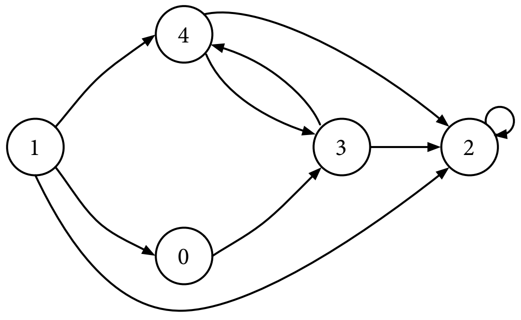

Figure 10.2: Two representations of the same directed graph.

Example solutions

One way to specify a general graph is a simple two-dimensional table, where each entry represents a directed link between the start node (row number) and the end node (column number) (see Figure 10.2). One simple way to sonify such a graph would be to move from row to row and assign to each node X an n-dimensional sound event (e.g., a frequency spectrum) according to the set of nodes $S = \{X_0, X_1, \ldots X_n\}$ to which it is connected. This would sonify all links (vertices) S of each node. In order to also hear which node the connections belong to, the first example simply plays the starting node, then its connections, and the next node is separated by a longer pause (hear sound example **S10.20**).

```
q = ();

SynthDef(\x, { |freq = 440, amp = 0.1, sustain = 1.0, out = 0|
    var signal = SinOsc.ar(freq);
    var env = EnvGen.kr(Env.perc(0.01, sustain, amp), doneAction: 2);
    Out.ar(out, signal * env);
}).add;

q.graph = [
    [0, 0, 0, 1, 0],
    [1, 0, 1, 0, 1],
    [0, 0, 1, 0, 0],
    [0, 0, 1, 0, 1],
    [0, 0, 1, 1, 0]
];
    // arbitrary set of pitches to label nodes:
q.nodeNotes = (0..4) * 2.4; // equal tempered pentatonic

Task {
        // iterating over all nodes (order defaults to listing order)
    loop {
        q.graph.do { |arrows, i|
            var basenote = q.nodeNotes[i];
                // find the indices of the connected nodes:
            var indices = arrows.collect { |x, i| if(x > 0,  i, nil) };
                // keep only the connected indices (remove nils)
            var connectedIndices = indices.select { |x| x.notNil };
                // look up their pitches/note values
            var connectedNotes = q.nodeNotes[connectedIndices];
            (instrument: \x, note: basenote).play;
            0.15.wait;

            (instrument: \x, note: connectedNotes).play;
            0.45.wait;
    };
    0.5.wait;
    };
}.play;
```

Another way of displaying the structure is to follow the unidirectional connections: each node plays, then its connections, then one of the connections is chosen as the next starting node. While the first example looked at the connections "from above", this procedure remains faithful to the locality of connections as they appear "from inside" the graph (hear sound example **S10.21**).

```
Task {
    var playAndChoose = { |nodeIndex = 0|
        var indices = q.graph[nodeIndex].collect { |x, i| if(x > 0,  i, nil) };
        var connectedIndices = indices.select { |x| x.notNil };

        var basenote = q.nodeNotes[nodeIndex].postln;
        var connectedNotes = q.nodeNotes[connectedIndices];

        (instrument: \x, note: basenote).play;
        0.15.wait;

        (instrument: \x, note: connectedNotes).play;
        0.3.wait;
            // pick one connection and follow it
        playAndChoose.value(connectedIndices.choose);
    };
    playAndChoose.value(q.size.rand); // start with a random node
}.play;
```

As node 2 only points to itself, the sonification quickly converges on a single sound that only connects to itself. If one changes node 2 to have more connections, other nodes become accessible again.

```
    // connect node node 2 to node 4:
q.graph[2].put(4, 1); // nodes 3 and 4 become accessible
q.graph[2].put(4, 0); // disconnect 4 from 2 again

q.graph[2].put(1, 1); // connect 2 to 1: all nodes are accessible now
q.graph[2].put(1, 0);  // disconnect 1 from 2 again
```

Discussion

The examples given here were chosen for simplicity; certainly, more complex strategies can be imagined.

The last design discussed above could be extended by supporting *weighted connections* – they could be mapped to amplitude of the sounds representing the connected nodes, and to probability weights for choosing the next node among the available connections. One would likely want to include a halting condition for when a node has no further connections, maybe by introducing a longer pause, then beginning again with a randomly picked new node. These additions would allow monitoring a graph where connections gradually evolve over time. Adding information to the edge itself (here, a real number between 0 and 1) is one case of a *labeled graph*. Finally, this can be considered a sonification of an algorithm (as discussed in the next section 10.4.5): such a graph is a specification of a finite state machine or its statistical relative, a Markov chain. In a similar way, the syntactic structure of a program, which forms a graph, may be sonified.

A *tree* is a special case of a graph in which every vertex has a specific level. So instead of adding extra information to the edges, also the vertex may be augmented. In the case of a tree, we augment each vertex by adding a partial order that tells us whether this node is on a higher, a lower, or on the same level as any other node that it is connected to. This is important wherever there is some hierarchy or clustering which we try to investigate. For sonification, this additional information can be used to specify the way the graph is traversed (e.g., starting from the highest level and going down to the lowest first – *depth-first*, or covering every level first – *breadth-first*). Note that wherever one wants to guarantee that every edge is only sonified once, it is necessary to keep a list of edges already traversed. The order information need not solely inform the time order of sound events but also other parameters, such as pitch (see section 10.4.2). Also it is possible to sonify a graph without traversing it over time – it can also serve as a model for a synthesis tree directly [6].

So far we have discussed only directed and connected graphs. Undirected graphs may be represented by directed ones in which each pair of vertices is connected by two edges, so they don't pose new difficulties as such. For graphs that consist of several separate parts, we can first fully traverse all paths to find which parts exist. Then each subgraph can be sonified. Note that the first example, which uses the table of connections directly, works the same way with connected and unconnected graphs.

In general, graph traversal is a well covered problem in computer science, whose results offer a wide range of possibilities for graph sonification, most of which go far beyond anything covered here.

10.4.5 Algorithms: Sonifying causal and logical relations

There are cases in which we can represent the result of a process as a simple sequence of data points, but if we are interested in conditions and causality of processes, these often become part of what we want to sonify. So far, such causal and logical relations have only been implicit in the sound (in so far as they may become audible as a result of a successful sonification), but not explicit in the structure of the experiment. This is the next step to be taken, and next is shown how the sonification laboratory may provide methods to bring to the foreground the causal or logical relations between data.

By definition, within a computational system, causal relations are algorithmic relations and causal processes are computational processes. In a broad understanding, algorithms make up a reactive or interactive system, which can be described by a formal language. We may also take algorithms simply as systematic patterns of action. In a more narrow understanding, algorithms translate inputs via finite steps into definite outcomes.[15] Generally, we can say that – to the same degree that *cause and effect* are intertwined – algorithms connect one state and with the other in a specific manner. In such a way, they may serve as a way to represent natural laws, or definite relations between events. If we know how to sonify algorithms we have at our disposal the means to sonify data together with their theoretical context.

Up to this point we have already implicitly sonified algorithms: we have used algorithms to sonify data – they represented something like the transparent medium in which the relation between measured data points became apparent. It is this medium itself which becomes central now. This may happen on different levels.

The algorithm itself may be sonified:

1. in terms of its output (we call this *effective sonification* – treating it as a black box, equivalent algorithms are the same),
2. in terms of its internal steps (we call this *procedural sonification* – equivalent algorithms are different if they proceed differently),
3. in terms of the structure of its formal description (see section 10.4.4).

While often intertwined, we may think of different reasons why such a sonification may be significant:

1. we are interested in its mathematical properties,

[15]The term algorithm is ambiguous. For a useful discussion, see for instance [10].

2. it represents some of the assumed causal or logical chains between states of a system,
3. it reproduces some of the expected effects (simulation).

Examples for procedural and effective sonification of algorithms

For a demonstration of the difference between the first two approaches, effective and procedural sonification, there now follows a very simple example of the Euclidean Algorithm, which is still today an effective way to calculate the greatest common divisor of two whole numbers. For this we need only to repeatedly subtract the smaller number from the larger number or, slightly faster, obtain the rest of integer division (modulo).

In order to hear the relation between the input of the algorithm and its output, we sonify both the pair of its operands and the greatest common divisor (gcd) thus obtained as a chord of sine tones. We may call this *effective sonification* of an algorithm. As we only sonify its outcome, it should sound the same for different versions and even other algorithms that solve the same problem.

Two sets of numbers are provided below for whose pairs the gcd is calculated. We use a random set here. Each pair is presented together with its gcd as a chord of sine tones whose frequencies in Hertz are derived from a simple mapping function $g(x) = 100x$, whose offset guarantees that the lowest value $x = 1$ corresponds to an audible frequency (100 Hz) (hear sound example **S10.22**).

```
f = { |a, b|
    var t;
    while {
        b != 0
    } {
        t = b;
        b = a mod: b;
        a = t;
    };
    a
};

g = { |i| i * 100 }; // define a mapping from  natural numbers to frequencies.

SynthDef(\x, { |freq = 440, amp = 0.1, sustain = 1.0, out = 0|
    var signal = SinOsc.ar(freq) * AmpComp.ir(freq.max(50));
    var env = EnvGen.kr(Env.perc(0.01, sustain, amp), doneAction: 2);
    Out.ar(out, signal * env);
}).add;

Task {
    var n = 64;
    var a = { rrand (1, 100) } ! n; // a set n random numbers <= 100
    var b = { rrand (1, 100) } ! n; // and a second dataset.
    n.do { |i|
        var x = a[i], y = b[i];
        var gcd = f.value(x, y);
        var nums = [x, y, gcd].postln; // two operands and the result
        var freqs = g.value(nums);    // mapped to 3 freqs ...
                        // in a chord of a sine grains
             (instrument: \x, freq: freqs).play;
             0.1.wait;
    }

}.play
```

To realise a *procedural implementation* of the same algorithm, we have to access its internal variables. One way to do this is to evaluate a *callback function* at each iteration, passing the intermediate values back to where the gcd algorithm was called from. Within a co-routine like `Task`, the function may halt the algorithm for a moment in the middle (here 0.1 s), sonifying the intermediate steps. In our example, the intermediate gcd values are sonified as a pair of sine tones (hear sound example **S10.23**).

```
f = { |a, b, func|
    var t;
    while {
        b != 0
    } {        // return values before b can become 0
        func.value(a, b, t);
        t = b;
        b = a mod: b;
        a = t;
    };
};

// procedural sonification of the Euclidean algorithm
Task {
    var n = 64;
    var a = { rrand (1, 100) } ! n; // n random numbers <= 100.
    var b = { rrand (1, 100) } ! n; // and a second dataset.
    n.do { |i|
            f.value(a[i], b[i], { |a, b| // pass the
            var numbers = [a, b].postln;
                // a 2 note chord of sine grains
                 (instrument: \x, freq: g.value(numbers)).play;
                 0.1.wait; // halt briefly after each step
            });
            0.5.wait; // longer pause after each pair of operands
    }
}.play;
```

Operator based sonification

For the sonification laboratory it is essential to enable the researcher to easily move the border between measured data and theoretical background, so that tacit assumptions about either become evident. Integrating data and theory may help to develop both empirical data collection and the assumed laws that cause coherence. A sonification of a physical law, for instance, may be done by integrating the formal relations between entities into the sound generation algorithm (because this effectively maps a domain function to a sonification function, we call this *operator based sonification* [26, 30]).

An object falling from great height can, for simplicity, be sonified by assigning the height h to the frequency of a sine tone (assuming no air resistance and other effects): $y(t) = \sin(2\pi\theta t)$, where the phase $\theta = \int(h_0 - gt^2)dt$. For heights below 40 m, however, this sine wave is inaudible to the human ear ($f < 40$ Hz). Also, dependent on gravity, the fall may be too short. The sonification introduces scalings for appropriate parameter mapping (see section 10.4.2), changing the rate of change (duration) and the scaling of the sine frequency (k) (hear sound example **S10.24**).

```

(
SynthDef(\fall, { |h0 = 30, duration = 3, freqScale = 30|
    var y, law, integral, g, t, h, freq, phase, k;
    g = 9.81; // gravity constant
    t = Line.ar(0, duration, duration); // advancing time (sec)
    law = { |t| g * t.squared }; // Newtonian free fall
    integral = { |x| Integrator.ar(x) * SampleDur.ir };
    h = h0 - law.value(t); // changing height
    freq = (max(h, 0) * freqScale); // stop at bottom, scale
    phase = integral.(freq); // calculate sin phase
    y = sin(2pi * phase);
            // output sound - envelope frees synth when done
    Out.ar(0, y * Linen.kr(h > 0, releaseTime:0.1, doneAction:2));
  }).add;
);

Synth(\fall);

```

This however causes an ambiguity between values belonging to the sound algorithm $\sin(k2\pi t f)$ and those belonging to the sonification domain $h_0 - gt^2$. This simple example does not pose many problems, but for more complex attempts, it can be crucial to formally separate the domains more clearly.

This problem can be addressed by introducing *sonification variables* into the formalism that usually describes the domain. By superscribing variables that belong to the context of sonification by a *ring*,[16] sonification time is therefore distinguished as $\mathring{t}$ from the domain time variable t, and the audio signal y itself can be similarly marked as $\mathring{y}$. The above example's semantics become clearer: $\mathring{y}(\mathring{t}) = \sin(\mathring{k}2\pi\mathring{t} \int (h_0 - g\mathring{t})^2 d\mathring{t})$. All those variables which are introduced by the sonification are distinguishable, while all terms remain entirely explicit and do not lose their physical interpretation. For a discussion of sonification variables and operator based sonification, especially from quantum mechanics, see [30].

Integrating data and theory

For demonstrating how to combine this sonification of a physical law with experimental data, take a classical example in physics, namely Galileo Galilei's inclined plane experiment (from a fragment from 1604), which is classical also in the historiography of sonification. Before Stillman Drake's publications in the 1970s [13], it was assumed that Galileo measured time by means of a water clock. In a previously lost document, Drake surprisingly discovered evidence for a very different method. According to Drake, Galileo adjusted moveable gut frets on the inclined plane so that the ball touched them on its way down. These "detectors" could be moved until a regular rhythm could be heard, despite the accelerating motion of the ball.[17] This experiment has been reconstructed in various versions for didactic purposes, as well as for historical confirmation [25], [3], partly using adjustable strings or bells instead of gut frets.

The code below shows a simulation of the inclined plane experiment, in which the law of gravity ($s = gt^2$), only stated in this form later by Newton, is assumed. A list of distances (*pointsOfTouch*) is given at which the "detectors" are attached. Time is mapped as a linear parameter to the distance of the accelerating ball, and whenever this distance exceeds the

[16] In LaTeX, the little ring is written as \mathring{. . . }

[17] Drake's [12] more general discussion of Renaissance music provokes the idea of a possible continuity in the material culture of Mediterranean laboratory equipment: in a sense, this experiment is a remote relative of the Pythagorean monochord – just as tinkering with the latter allowed the discovery of an invariance in frequency ratios, the former helped to show the invariance in the law of gravity. At the time, many artists and theorists were Neopythagoreans, like Galileo's father [16].

distance of one of the detectors, it is triggered (hear sound example **S10.25**).

```

(
Ndef(\x, {
    var law, g = 9.81, angle;
    var pointsOfTouch;
    var ball, time, sound, grid;

//  pointsOfTouch = [1, 2, 3, 5, 8, 13, 21, 34]; // a wrong estimate
        // typical measured points by Riess et al (multiples of 3.1 cm):
    pointsOfTouch = [1, 4, 9, 16.1, 25.4, 35.5, 48.5, 63.7] * 0.031;

    angle = 1.9; // inclination of the plane in degrees
    law = { |t, gravity, angle|
        sin(angle / 360 * 2pi) * gravity * squared(t)
    };

    // linear "procession" of time:
    time = Line.ar(0, 60, 60);
        // distance of ball from origin is a function of time:
    ball = law.value(time, g, angle);

    sound = pointsOfTouch.collect { |distance, i|
        var passedPoint = ball > distance; // 0.0 if false, 1.0 if true
            // HPZ2: only a change from 0.0 to 1.0 triggers
        var trigger = HPZ2.ar(passedPoint);
            // simulate the ball hitting each gut fret
        Klank.ar(
        `[
            {exprand(100, 500) }    ! 5,
            { 1.0.rand }            ! 5,
            { exprand(0.02, 0.04) } ! 5
        ],
        Decay2.ar(trigger, 0.001, 0.01, PinkNoise.ar(1))
       )
    };

        // distribute points of touch in the stereo field from left to right
    Splay.ar(sound) * 10
        // optionally, add an acoustic reference grid
//  + Ringz.ar(HPZ2.ar(ball % (1/4)), 5000, 0.01);
}).play
)
```

A slightly richer, but of course historically less accurate variant of the above uses the method employed by Riess et al., in which it is not the frets that detect the rolling ball, but instrument strings (hear sound example **S10.26**).

```
Ndef(\x, {
    var law, g = 9.81, angle;
    var pointsOfTouch;
    var ball, time, sound, grid;

//  pointsOfTouch = [1, 2, 3, 5, 8, 13, 21, 34]; // a wrong estimate
        // typical measured points by Riess et al (multiples of 3.1 cm):
    pointsOfTouch = [1, 4, 9, 16.1, 25.4, 35.5, 48.5, 63.7] * 0.031;

    angle = 1.9; // inclination of the plane in degrees
    law = { |t, gravity, angle|
        sin(angle / 360 * 2pi) * gravity * squared(t)
    };

    // linear "procession" of time:
    time = Line.ar(0, 60, 60);
        // distance of ball from origin is a function of time:
    ball = law.value(time, g, angle);

    sound = pointsOfTouch.collect { |distance, i|
        var passedPoint = ball > distance; // 0.0 if false, 1.0 if true
            // HPZ2: only a change from 0.0 to 1.0 triggers
        var trigger = HPZ2.ar(passedPoint);
            // using Galileo's father's music theory for tone intervals
        var freq = 1040 * ((17/18) ** i);
            // simple vibrating string model by comb filter.
        CombL.ar(trigger, 0.1, 1/freq, 1.0 + 0.3.rand2)
    };

        // distribute points of touch in the stereo field from left to right
    Splay.ar(sound) * 10
        // optionally, add an acoustic reference grid
//  + Ringz.ar(HPZ2.ar(ball % (1/4)), 5000, 0.01);
}).play
```

Discussion

Evaluating the code above with varying settings for *pointsOfTouch*, and various angles, one can get an impression of the kind of precision possible in this setup. In a sense, it is much better suited than a visual reconstruction, because there are no visual clues that could distract the listening researcher. The whole example may serve as a starting point for quite different sonifications: it demonstrates how to acoustically relate a set of points with a continuous function. Replacing the *law* by another function would be a first step in the direction of an entirely different model.

As soon as one becomes aware of the fact that we do not only discover correlations in data, but also tacitly presume them – the border between auditory display and theory turns out to be porous. Correlations may either hint toward a causality in the domain or simply be a consequence of an artefact of sonification. Integrating data and theory more explicitly helps experimentation with this delimitation and the adjustment of it so that the sonification displays not only itself. The introduction of sonification variables may help both with a better understanding of the given tacit assumptions and with the task of finding a common language between disciplines, such as physics, mathematics, and sonification research.

As we have seen, one and the same algorithm gives rise to many different perspectives of sonification. Separating its "outside" (its effect) from its "inside" (its structure and procedural behavior) showed two extremes in this intricate spectrum.

Finally, this presentation gives a hint of an interesting way to approach the question *"What do we hear?"*. Necessarily, we hear a mix of the sonification method and its domain in every instance. Within the series of sound events in Galileo's experiment, for instance, we listen to the device (the frets and their arrangement) just as much as we listen to the law of gravity that determines the movement of the accelerating ball. Sonification research is interested in distal cues, outside of its own apparatus (which also produces proximal cues: see section 10.4). We try to hear the domain "through" the method, so to speak. There is an inside and an outside also to sonification.

Furthermore, data itself may be considered the external effect of an underlying hidden logic or causality, which is the actual subject of investigation. Unless surface data observation is sufficient for the task at hand, the "outside" of sonification is also the "inside" of the domain we are investigating.

Add to this the fact that today's sonification laboratory consists almost exclusively of algorithms, which may either be motivated by the sonic method or by the domain theory. Which "outside" are we finally listening to? How is the structure of the algorithm tied to the structure of the phenomenon displayed? What we face here is a veritable epistemological knot. This knot is implicit in the practice of sonification research, and each meaningful auditory display resolves it in one way or another.

10.5 Coda: back to the drawing board

This chapter has described a range of methods considered essential for a sonification laboratory, introduced SuperCollider, a programming language that is well suited for sonification research in lab conditions, and suggested guidelines for working on sonification designs and their implementations. This should provide useful orientation and context for developing appropriate methods for the acoustic perceptualization of knowledge.

However, depending on the domain under exploration, and the data concerned and its structures, each problem may need adaptations, or even the invention of new methods. This means a repeated return to the drawing board, revising not only the data, but also the equipment – calibrating sonification methods, programming interfaces, and discussing the implications of both approach and results. As sonification research involves multiple domains and communities, it is essential that all participants develop a vocabulary for cross-disciplinary communication and exchange; otherwise, the research effort will be less effective.

A sonification laboratory, being an ecosystem situated between and across various disciplines, should be capable of being both extremely precise and allowing leeway for rough sketching and productive errors. It is precisely this half-controlled continuum between purity and dirt [11] that makes a laboratory a place for discoveries. Doing sonification research means dealing with large numbers of notes, scribbles, software and implementation versions, incompatible data formats, and (potentially creative) misunderstandings. The clarity of a result is not usually present at the outset of this process – typically, the distinction between fact and artefact happens along the way.

Finally, a note on publishing, archiving and maintaining results: science thrives on generous open access to information, and the rate of change of computer technology constantly endangers useful working implementations. Sonification laboratory research does well to address both issues by adopting the traditions of open source software and literate programming. Publishing one's results along with the entire code, and documenting that code so clearly that re-implementations in new contexts become not just possible but actually practical makes one's research contributions much more valuable to the community, and will quite likely increase their lifetime considerably.

Bibliography

[1] G. Bachelard. *The Formation of the Scientific Mind. A Contribution to a Psychoanalysis of Objective Knowledge*. Clinamen Press, 2002 [1938].

[2] S. Barrass. *Auditory Information Design*. PhD thesis, Australian National University, 1997.

[3] F. Bevilacqua, G. Bonera, L. Borghi, A.D. Ambrosis, and CI Massara. Computer simulation and historical experiments. *European Journal of Physics*, 11:15, 1990.

[4] R. Boulanger. *The Csound Book: Perspectives in Software Synthesis, Sound Design, Signal Processing, and Programming*. MIT Press, Cambridge, MA, USA, 2000.

[5] T. Bovermann. *Tangible Auditory Interfaces. Combining Auditory Displays and Tangible Interfaces*. PhD thesis, Bielefeld University, 2009.

[6] T. Bovermann, J. Rohrhuber, and H. Ritter. Durcheinander. understanding clustering via interactive sonification. In *Proceedings of the 14th International Conference on Auditory Display*, Paris, France, June 2008.

[7] R. Candey, A. Schertenleib, and W. Diaz, Merced. xSonify: Sonification Tool for Space Physics. In *Proc. Int Conf. on Auditory Display (ICAD)*, London, UK, 2006.

[8] A. de Campo. Toward a Sonification Design Space Map. In *Proc. Int Conf. on Auditory Display (ICAD)*, Montreal, Canada, 2007.

[9] A. de Campo. *Science By Ear. An Interdisciplinary Approach to Sonifying Scientific Data*. PhD thesis, University for Music and Dramatic Arts Graz, Graz, Austria, 2009.

[10] E. Dietrich. Algorithm. In The MIT Encyclopedia of the Cognitive Sciences (Bradford Book), 1999.

[11] M. Douglas. *Purity and Danger*. Routledge and Kegan Paul, London, 1966.

[12] S. Drake. Renaissance music and experimental science. *Journal of the History of Ideas*, 31(4):483–500, 1970.

[13] S. Drake. The Role of Music in Galileo's Experiments. *Scientific American*, 232(8):98–104, June 1975.

[14] P. Galison. *Image & logic: A material culture of microphysics.* The University of Chicago Press, Chicago, 1997.

[15] T. Hermann and H. Ritter. Listen to your Data: Model-Based Sonification for Data Analysis. In *Advances in intelligent computing and multimedia systems*, pages 189–194, Baden-Baden, Germany, 1999. Int. Inst. for Advanced Studies in System research and cybernetics.

[16] C. Huffman. Pythagoreanism. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Summer 2010 edition, 2010.

[17] D. E. Knuth. *Literate Programming*. CSLI Lecture Notes, no. 27. Center for the Study of Language and Information, Stanford, California, 1992.

[18] M. Mathews and J. Miller. *Music IV programmer's manual.* Bell Telephone Laboratories, Murray Hill, NJ, USA, 1963.

[19] J. McCartney. Supercollider: A new real-time synthesis language. In *Proc. ICMC*, 1996.

[20] J. McCartney. Rethinking the computer music language: Supercollider. *Computer Music Journal*, 26(4):61–68, 2002.

[21] S. Pauletto and A. Hunt. A Toolkit for Interactive Sonification. In *Proceedings of ICAD 2004, Sydney*, 2004.

[22] S. Pauletto and A. Hunt. The Sonification of EMG data. In *Proceedings of the International Conference on Auditory Display (ICAD)*, London, UK, 2006.

[23] J. Piché and A. Burton. Cecilia: A Production Interface to Csound. *Computer Music Journal*, 22(2):52–55, 1998.

[24] H.-J. Rheinberger. *Experimentalsysteme und Epistemische Dinge (Experimental Systems and Epistemic Things).* Suhrkamp, Germany, 2006.

[25] F. Riess, P. Heering, and D. Nawrath. Reconstructing Galileos Inclined Plane Experiments for Teaching Purposes. In *Proceedings of the International History, Philosophy, Sociology and Science Teaching Conference*, Leeds, 2005.

[26] J. Rohrhuber. $\mathring{\mathcal{S}}$ – Introducing sonification variables . In *Proceedings of the Supercollider Symposium*, Berlin, 2010.

[27] A. Schoon and F. Dombois. Sonification in Music. In *International Conference on Auditory Display*, Copenhagen, 2009.

[28] B. Snyder. *Music and Memory.* MIT Press, 2000.

[29] B. Vercoe. *CSOUND: A Manual for the Audio Processing System and Supporting Programs.* M.I.T. Media

Laboratory, Cambridge, MA, USA, 1986.

[30] K. Vogt. *Sonification of Simulations in Computational Physics*. PhD thesis, Institute for Physics, Department of Theoretical Physics, University of Graz, Austria, Graz, 2010.

[31] B. Walker and J. T. Cothran. Sonification Sandbox: A Graphical Toolkit for Auditory Graphs. In *Proceedings of ICAD 2003, Boston*, 2003.

[32] S. Wilson, D. Cottle, and N. Collins, editors. *The SuperCollider Book*. MIT Press, Cambridge, MA, 2011.

[33] D. Worrall, M. Bylstra, S. Barrass, and R. Dean. SoniPy: The Design of an Extendable Software Framework for Sonification Research and Auditory Display. In *Proc. Int Conf. on Auditory Display (ICAD)*, Montreal, Canada, 2007.

[34] M. Wright and A. Freed. Open sound control: A new protocol for communicating with sound synthesizers. In *International Computer Music Conference*, pages 101–104, Thessaloniki, Hellas, 1997. International Computer Music Association.