Chapter 5

THE *SONIPY* SOFTWARE FRAMEWORK FOR DATA SONIFICATION

The need for better software tools was highlighted in 1979 in a comprehensive status review of the field, published as the *Sonification Report* (Kramer et al. 1999). It included some general proposals for adapting sound synthesis software to the needs of sonification research. Now, over a decade later, it is evident that the demands on software by sonification research are greater than those afforded by music composition and sound synthesis software. This chapter compares some major contributions made towards achieving the *Report*'s proposals with current sonification demands and outlines *SoniPy*, a broader and more robust framework model that can integrate the expertise and prior development of software components using a public-domain community-development approach.

5.1 Introduction

To date, software for data sonification has been developed either as standalone applications engineered from first principles, sometimes incorporating third-party low-level audio routines, or as more expansive sonification 'environments' that attempt to encapsulate some general principles and procedures that can be adapted for specific sonification projects as the need arises.

The standalone applications tend to be designed for individual experiments entailing clearly defined tasks such as accurate monitoring (Chafe and Leistikow 2001), or graphic user-interaction (Walker and Cothran 2003), whereas environments tend to be more expansive projects, often with less deterministic outcomes. They afford greater flexibility than is possible within standalone applications. Some recent environments still in development (Pauletto and Hunt 2004; de Campo, Frauenberger and Höldrich 2004) seem to have been designed by first choosing a music composition environment and working backwards, perhaps trusting that the data-processing needs at the 'input end' can be adequately handled by the language tools available from within the particular composition system

chosen. Considering that many software tools for music composition have a long gestation period (in the case of *Csound*[1], about forty years, for example) and are still being actively developed, as can be witnessed, for example, by the daily activity on the developer mailing lists for *Csound* and *SuperCollider*[2], this approach is natural and is the approach assumed in the *Sonification Report*.

In addition to the two primarily scripted environments just mentioned, *MAX/MSP*[3] and its younger sibling *PD*[4] are another type, which emphasises graphical user interfaces (GUIs) over program code. Whilst the scripting-versus-GUI debate is still active, it is clear from the large user-base and active development of new patch objects for these platforms that the GUI approach is appealing to some users and perhaps offers a gentler initial learning curve for many exploratory sonification researchers who are visually inclined. In any event, a considerable investment of time is necessary to become proficient in any of these environments and having made the investment, a certain amount of environment 'stickiness' is apparent and understandable.

5.1.1 The First Major Bottleneck: Data

In data sonification, whilst the input data can be thought of as eventually controlling the sound rendering, the transformations it has to undergo in the interim can be considerable. Such data processing can reasonably include multidimensional scaling, filtering and statistical analysis which itself may itself become the subject of sonification. Also, each input dataset can have potentially unique structural characteristics. Some, such as EEG data, may be multiple channels of AC voltages with a variety of DC-biases and noisiness as determined by the particular data collection set-up on a particular patient. Others, such as security data flowing from a market trading-engine, will be massively paralleled, metadata embedded and multiplexed into a single 'feed'. Difficulties in using such data are compounded when the it needs to be buffered and streamed in non-real-time as is the need for multiple overlays of time sequences of different temporal compressions.

---

1 http://www.counds.com/
2 http://www.audiosynth.com/
3 http://www.cycling74.com/
4 http://crca.ucsd.edu/~msp/software.html/

High-level tools for processing such data complexities are rarely, if ever, found in computer music environments, and even less likely if the input data is spatial rather than temporal. When such an a environment is the principle sonification tool, a common response to complex data processing requirements is for someone, where possible, to 'bite the bullet' and write data-processing routines in the language of the composition environment itself. This is currently the approach used by *SonEnvir* (de Campo Frauenberger and Höldrich 2004) and *OctaveSC*[5] (Hermann et al. 2006), which both use *SuperCollider* and also the *PD*-based *Interactive Sonification Toolkit* (Pauletto and Hunt 2004)[6]. The following is a recent exchange on the *Supercollider* users list which epitomises the situation exactly:

```
Question: Hi, I'm looking for linear algebra and affine
transformation routines for 2D and 3D vector spaces; is there
any quark or extension that implement stuff like that?
Answer: The MathLib quark[7] maybe has some useful stuff... or it
would be the place to put these.
```

…or it would be the place to put these implies if someone else using *SCLang* hasn't erstwhile written them and made them publicly available. The user has to make a decision as to whether or not they have the required expertise and time to dedicate to implementing them. While *SClang* is a very elegant and powerful composition language that *can* support the development of data-processing solutions, being unique, it lacks the transportability that more general and widely available tools afford. One consequence of this is that, in projects without a dedicated programmer, practical assistance for what are essentially data processing problems is more difficult to obtain. Data is thus often pre-processed using external tools such as spreadsheets and then read from files by the music composition environment; a procedure that, whilst it may be appropriate in 'limited data' experiments, is at best susceptible to data corruption and of no use if the data is coming from a real-time feed or from a dynamic model (Bovermann, Hermann, and Ritter 2006).

This situation may be characterised as 'data SONIFICATION' i.e. the primary focus is on sound rendering whilst input data is constrained so it can be dealt with adequately by the rendering software. The alternative, an emphasis on data-processing tools at the expense of sound rendering

[5] http://www.sonification.de/projects/sc3/index.shtml/
[6] PD is Miller Puckette's public release of his Max/MSP.
[7] A Quark is a package of SC classes, helpfiles, C++ sourcecode for unit generators and other code.

flexibility ('DATA sonification') is no more attractive because the sound palette tends to be small and the range of controls limited, the outcome of which is too-often difficult to listen to over extended periods of time. Some examples of this latter approach include extensions for the *Matlab* numerical environment (Miele 2003), *AVS Visualization Toolkit* (Kaplan, B. 1993), *Excel* spreadsheets (Stockman, Hind and Frauenberger 2005), and the *R* statistical analysis package (Heymann and Hansen 2002). They provide data handling and processing capabilities but very basic sample-based sound capabilities modelled on the MIDI protocol. What are needed are tools that afford a balanced, equally-flexible approach.

Excellent data-processing tools exist in the public domain (see §5.3.2.2 below) and are an integral part of much scientific research. Furthermore, they are continually being extended and modernised by teams of developers spread across the world. Yet because of the decision to use a music composition environment for sound rendering, these tools remain inaccessible to most sonification environments. Whether this situation has come about because the data for music composition by computer is mostly internally generated rather than externally acquired is open to debate. However, until sophisticated tools for handling externally-acquired often massively multiplexed datasets can be brought to bear on the acquisition, analysis, storage and re-presentation requirements of the sonification process, even before any mapping or sound rendering is undertaken, there is limited chance that such software will enable the *choosing* [of] *mappings between variables and sounds interactively, navigating through the data set, synchronizing the sonic output with other display media, and performing standard psychophysical experiments* that the *Sonification Report* envisaged.

5.1.2 Motivations

Many of the issues raised in the previous section are generic to software in general. Anecdotally, 'late delivery' and 'over-budget' are as common as they are notorious characteristics of the commercial software industry as all but the largest applications find it difficult to maintain hardware and operating-system currency; two of the essential activities necessary to meeting their obligations to paying customers, if they hope to survive. Some do, of course,

but increasingly developers and users are turning to open-source, community support, to sustain ongoing viability in the medium-to-longer-term.

The need for a serious solution to the issues outlined above was first undertaken in the context of this thesis as a result of the difficulties encountered in trying to sonify a single large multidimensional dataset for the ICAD-04 *Listening to the Mind Listening* project. The sonifiers involved used different hardware platforms, under different versions of operating systems, and with each having a preference for, and expertise in a different collection of sound synthesis/music composition programs (Barrass 2004; Barrass, Whitelaw and Potard. 2005; Dean, Worrall and White 2004). While all involved technically literate, it was soon realised that if the difficulties experienced were any indication, it must be very difficult for almost everyone to achieve consistent, repeatable results under the same conditions with anything but the simplest datasets. This led us to specifying the requirements for an experimental software sonification framework. Some requirements are clearly identified in the *Sonification Report*, others of our own concoction. We call the framework *SoniPy*, in-keeping with the naming convention used for frameworks that extend the *Python* programming language.

5.2 *SoniPy*: concepts and requirements

SoniPy is designed to be a heterogeneous software framework for data sonification research and auditory display. It integrates various existing independent components such as those for data acquisition, storage and analysis, cognitive and perceptual mappings as well as sound synthesis and control, as illustrated in Figure 5.1, by encapsulating them, or control of them, as *Python* modules. The choice of *Python* was not arbitrary; it possesses all the features of a modern modular programming language that we consider essential for an experimental development environment. *Python* is

> a general-purpose programming language … which may also serve as a glue language connecting many separate software components in a simple and flexible manner, or as a steering language where high-level Python control modules guide low-level operations implemented by subroutine libraries effected in other languages. (Watter, van Rossen and Ahlstrom 1996).

Other descriptors include: simple, but not at the expense of expressive power, extensible, embeddable, interpreted, object-oriented, dynamically typed, upwardly compatible, portable and widely and freely available[8].

5.2.1 Design Requirements

As the *Sonification Report*'s *Sample Research Proposal* (op.cit. #3) acknowledges, the development of a comprehensive 'sonification shell' is not easy. The depth and breadth of knowledge, and skills required to effect such a project are easily underestimated. Whilst it has been a decade since the *Report* was published, progress has been quite slow. This is not to criticise those that have fallen by the wayside, nor those still in development, but to acknowledge both the difficulties involved in such a project and the need for new requirements if such projects are to have a better chance of survival. We briefly address the requirements indicated in the *Sonification Report* and add some of our own.

- *Integrability*. As discussed earlier with regard to data, due consideration needs to be taken of the requirements of the various components of the sonification and experimentation process. As is the case with most interdisciplinary ventures, each contributing discipline brings its collection of tools, techniques and standards to the venture and they need to be synergistically integrated. A software environment needs to be chosen that supports this goal. It is for this reason we have chosen *Python*, which can be used to 'wrap' independent pieces of conformable software in such a way as to permit data to flow between them. We follow *Python* convention and call them Modules.
- *Flexibility*. Rather than try to be the 'killer application,' *SoniPy* aims to wrap (inherit, or be extended by) the best collection of Modules available to it. These Modules need to have no computational interdependencies, though conceptually they may be similar, thus ensuring that no one of them is indispensable. Each of these Modules has evolved independently, probably over a considerable period of time. Independent of *SoniPy*, they have their own ongoing development teams that extend and improve then

[8] http://www.python.org/

as well as adapting them to ever-changing hardware and software platforms.

- *Extensibility*. In the situation where no Module exists for a particular task, a new Module can be designed in the knowledge that it will fit seamlessly within the existing Modular framework. This implies all Modules need to be thread compliant.
- *Accessibility.* It is desirable that as many Modules as possible be known in their own right. This reduces learning overhead for all users and enables work that may have already been undertaken with those tools to be accommodated within the *SoniPy* framework.
- *Portability. SoniPy* needs to be able to be instantiated on all major platforms. Furthermore, it is desirable, in certain applications, for Modules to be instantiated on different machines, in different locations and networked together: that is, to be heterogeneous.
- *Availability.* To protect both authors and users, *SoniPy* needs to be freely available with a minimum of restrictions. There are numerous licensing flavours for public-domain software whose source-code is made generally available, as outlined by the GNU organisation[9]. Because *SoniPy* employs heterogeneous components, the license of each component carries through into *SoniPy* in a way that is standard practice in the software industry. The *SoniPy*-specific components will be issued under the General Public License Version 2,[10] thus encouraging the sharing and free exchange of these tools in the community at the same time as enabling restrictions to be applied for individual projects as confidentiality agreements demand. *SoniPy*'s sources and documentation can be freely downloaded from its Sourceforge Internet repository[11].
- *Durability. SoniPy* needs to *survive*. While survival can never be guaranteed, in complex projects such as this, maximum risk-mitigation is essential. *SoniPy* is unlikely to survive if it remains the effort of a very small group. Essential to this is the community involvement and support in ongoing improvement and development: the very conditions under which *SoniPy*'s independent modules have been, and continue to be,

---

9 http://www.gnu.org/philosophy/categories.html
10 http://www.gnu.org/copyleft/gpl.html
11 http://sonipy.sourceforge.net/

developed. This approach is not without its own issues, as those who are involved in public-domain projects can attest (Luke et al. 2004).

5.2.2 Integration Through Wrapping

Although *Python* comes with an extensive standard library and there is a good resource of external *Python* libraries, we are not limited to using *Python* libraries. A powerful feature of *Python* is its set of well-defined interfaces to other languages. Libraries written in most languages can be integrated through *Python* by 'extending' it [24]. The basic principle of *SoniPy* is to use *Python* to 'wrap' independent software that can be compiled with python-bindings in such a way that data can flow between them. Quite a few tools exist for the (semi-) automatic generation of Python bindings, such as the *Simplified Wrapper and Interface Generator* (SWIG)[12].

Some applications provide *Python Application Program Interface* (API) libraries; other applications need to have a *Python* API written in order to use it (Lutz 1996: 505). Although some others embed *Python*, either by bundling it as an interpreter or by invoking the Python interpreter installed on the user's system as a basic API (ibid.: 505). We mention embedding in this context because, whilst it may be useful in its own right, it does not provide the interface flexibility needed by *SoniPy*. *SoniPy* requires an application to provide *Python* bindings so that *Python* can be *extended* by the application.

5.3 The design of SoniPy

The *SoniPy* design specifies five module sets communicating over two different networks: the *SonipyDataNetwork* (SDN) and the *SonipyControlNetwork* (SCN). Modules are grouped according to their role in the data sonification process: Data Processing (DP), Conceptual Modelling (CM), Psychoacoustic Modelling (PM), Sound Rendering (SR) and Monitoring & Feedback (MF). Depending on the dictates of a particular project, modules in a set may be instantiated on different machines. A particular module set may be empty, i.e. contain no modules, or a particular module may belong to more than one module set.

[12] http://www.swig.org/

5.3.1 Inter-Module Communication: The Networks

SoniPy's modular design makes it well suited for the instantiation of all selected modules on a single processor or, in order to take advantage of the computing power that multiple CPUs and machines can afford, the distribution of modules over multiple CPUs, a LAN or the Internet. We are currently extending *Python* versions greater than 2.4 under Apple's latest *OSX* operating system, but workable alternatives will flow as the development team expands.

Python's platform independence should assist the instantiation of parallel implementation of *SoniPy*, distributed over a heterogeneous network. Some testing of different approaches to distributing the computing has been undertaken, in order to maximally benefit from the trade-off between performance (including real-time latency, data throughput and CPU overhead issues), ease-of-use, maintainability, reliability (over a network), scalability and heterogeneity; that is, the ability for non-*Python* third-party applications or devices to communicate with *SoniPy* modules (Coulouris, Dollimore and Kindberg 2005). Communication technologies being tested range from class inheritance, sockets, *OSC* and *MIDI* through to network audio mixing, using *Netjack* for example[13].

Other potential uses of a distributed approach include mobile phone sound-rendering and the processing of data remotely under local control, perhaps with the result being sent to another site for mapping, and psychoacoustic adjustment before being rendered to sound. The testing in ongoing and the interim results are not reliable enough to warrant reporting at the time of writing.

Referring to the diagrammatic representation of the way the module sets interrelate (Figure 5.1), it can be observed that *SoniPy*'s modules operate through two networks: *Data* and *Control*. The *SonipyDataNetwork* (SDN) is topologically configured as a bus whilst the *SonipyControlNetwork* (SCN) is a star configuration. This is analogous to the signal and control busses of an automated audio mixing desk. Control routing uses the same network technology as the data, though the destinations may be different. For example data from a DP module may be sent to a CM module on the SDN bus, under

---

13 http://netjack.sourceforge.net/

the control of an MF module communicating on the *SonipyControlNetwork*, without the data itself needing to go through an MF module. *Sonipy* controls need to be XML compliant and each module set may itself be the hub of a network of processors of topology unknown to the SCN router.

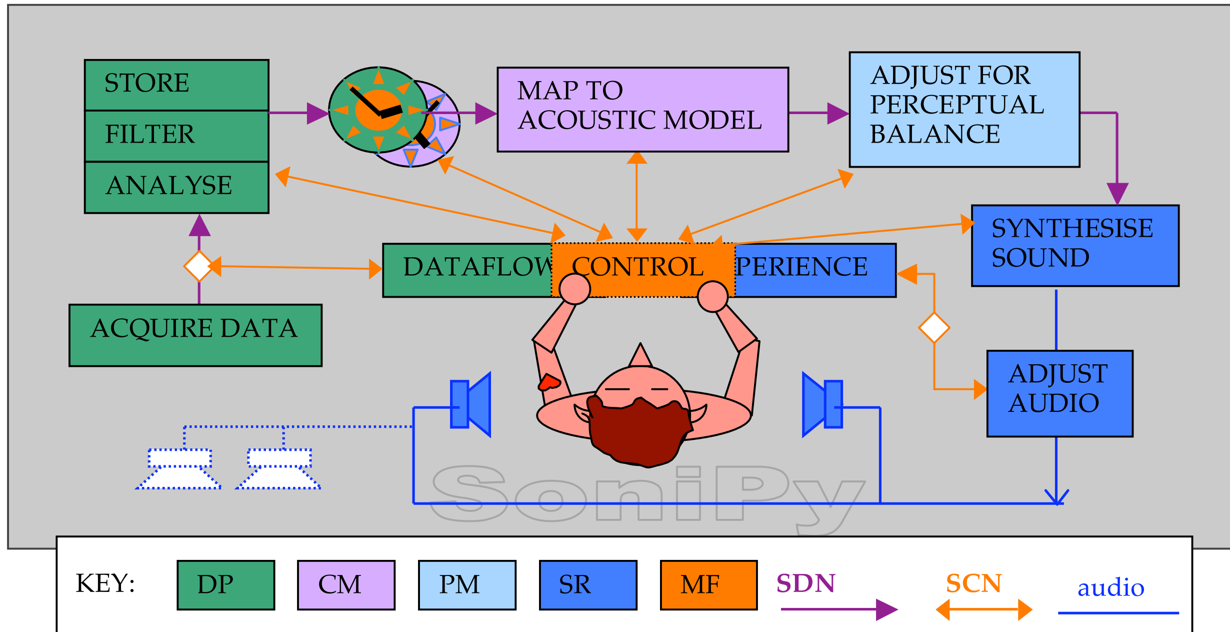

Figure 5.1. Flow diagram of *SoniPy*'s five module sets and two networks

Module Sets	**Data** **via SoniPy Data Network (SDN)**	**Controls** **via SoniPy Control Network (SCN)**
DP	Input: Raw data to be sonified Processes: Analysis, filtration, translation, storage Output: Modified Data to be sonified, Metadata	Process selection & OP switching options State (active, waiting, idle)
CM	Input: Data from DP, model selection algorithms Processes: Model Selection Output: Selected Model(s)	Process selection & output switching options State (active, waiting, idle)
PM	Input: Model control parameters Processes: Psychoacoustic transforms Output: Modified model control parameters	Process selection, & OP switch options Model instantiation map for render(s)) State (active, waiting, idle)
SR	Input: Synthesis models and controls Processes: Render sound Output: Sound	Audio controls: mute, unmute, gain etc Audio control changes State (active, waiting, idle)
MF	Input: Login, config. & process startup Processes: Initiate, route, record activity, monitor usage, system/networks state Output: UI, feedback, log of activity	State (active, waiting, idle, off) Resource usage, UI monitoring.

Table 5.1. Overview of some key features of *SoniPy* module sets . KEY: DP: Data Processing. CM: Conceptual Modelling, PM: Perceptual Modelling, SR: Sound Rendering, MF: Monitoring and Feedback, SDN: SoniPy Data Network, SCN: SoniPy Control Network.

5.3.2 The modules

Whenever possible, local modules are instantiated as library extensions to the *Python* programming framework. Should a module be instantiated remotely, that instantiation and the control of information to and from it remains under the control of the MF module that initiated the instantiation, where the main application loop as well as thread and GUI controls resides. Table 5.1provides an overview of some key features of the module sets.

5.3.2.1 Data processing modules

As the principal data processing activity in data sonification is taking place inside the listeners, the role of sonification is to prepare source data in a format that enables the listener to extract information, and in interactive systems, to take account of their feedback. *SoniPy*'s DP modules consist of a number of object-oriented classes which themselves inherit data classes and control classes according to the form and location of the raw data and its intended destination. Class methods include those for

- Interpolated lookup and mappings, for auditory icons and earcons, for example,
- Writing data to and extracting it from storage (memory, database and/or flat file) for pre-processing or multi-stream playback,
- Audification - writing data in formats acceptable as direct input to audio hardware,
- Simultaneous handling of multiple time-locked streams, such as from biomedical monitors,
- Deconstruction, analysis and filtering, including of complex meta-tag embedded multiplexed streams, such as a data feed from a stock-market trading engine,
- Model-based sonification involving user feedback, and
- Simulation of data feeds, including buffering with time compression and expansion.

Data and control class instances can be manipulated with *SciPy* (Scientific Python), a collection of open-source libraries for mathematics, science, and

engineering[14]. The core library is *NumPy*, which provides convenient and fast N-dimensional array manipulation. Figure 5.2 is a diagrammatic representation of one way of configuring *SoniPy*'s data processing modules under *SCN* control.

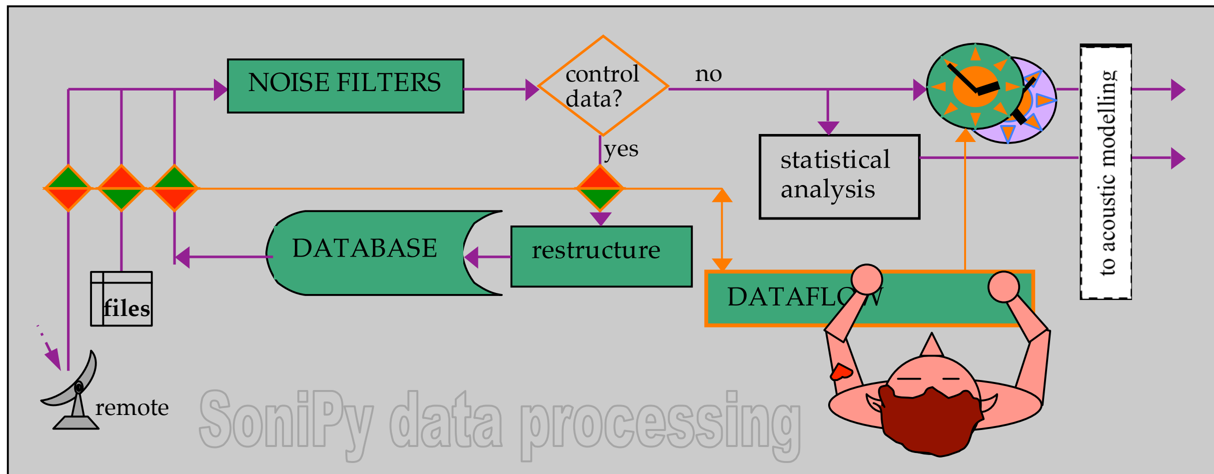

Figure 5.2. A map of the configuration of *SoniPy*'s Data Processing modules.

5.3.2.2 Scale, storage, access, and persistence

When the dataset used for sonification is finite, not too large, and is acquired in a timely manner, it can all be held in RAM and parsed using the basic data-manipulation tools of any modern programming language. Alternatively, as can be observed in the Auditory Display literature, it can be pre–processed separately using generic spreadsheet software. As the size, complexity and performance demands increase, as they frequently do multidimensional in real–time datasets, or when what is being sonified is some computed informational derivative of the raw data, the power and flexibility of an interpreter interface to a software framework reveal its superiority over purely music-oriented software[15].

In *Python*, direct access to many data management tools is available: Internal lists can be extended to typeless {key : value} dictionaries, then to flat files, perhaps with compression[16], and *NumPy* array processing, then using framework extension, to commercial relational databases like *Oracle* and the web-prevalent public domain *mySQL*. With very large datasets, such as those

---

[14] http://www.scipy.org/

[15] Including *SuperCollider*'s *scLang* probably the closest to Python in concept, in that it is text-based, interpretive and contains a dictionary class.

[16] Python has its own simple flat-file data compression module called *pickle*.

regularly employed in astronomy, meteorology and quantitative finance, one has to decide whether to use one of a number of models, such as *hierarchical*, *network*, *relational*, *entity-relationship*, *object-relational model* and *object*[17], depending on the structure of the data and the way it is used.

The relational model is characterized by relation, attribute, tuple, relation value and relation variable. A relation is, at its heart, a truth predicate about the world, a statement of facts (attributes) that provide meaning to the predicate (Date 2004). Speaking simply, relational systems are structured around relationships between facts and data retrieval is based on predicate logic and truth statements.[18] Objects systems are structured so as to emphasise identity, state, behaviour and encapsulation. An Object has a unique identity that is distinct from its state (the value of its internal fields)–two objects with the same state are still separate and distinct, despite being bit-for-bit mirrors of one another (Date 2004).

Whether to use an equivalence or identity model to structure data persistence[19] is a well–known programming problem: the object–relational–mapping– or ORM–dilemma.[20] The solution depends on both the structure and distribution of the data and the types of enquiry needing to be made of it, and can be the subject of extensive experimentation. Relational databases tend to perform more effectively when the data can be easily structured in a few relatively large tables in with a fixed number of fields and the object model when there are a large number of semi-autonomous instances. These issues are exemplified, with sample code, in the context of data sonification experiments using high-frequency capital market data in the next chapter (§5.8) where an HDF5–compliant b–tree structure was eventually employed using *pyTables*[21].

---

[17] Wikipedia has useful overviews. See http://en.wikipedia.org/wiki/Database_model and http://en.wikipedia.org/wiki/Object-relational_mapping.

[18] In essence, each row within a table is a location for a fact in the world, and a structured query language (SQL) allows for operator-efficient data retrieval of those facts using predicate logic to create inferences from them.

[19] In computer science, the term *persistence* refers to that characteristic of data that outlives the execution of the program that created it. Without persistence, data only exists in RAM, and will usually be 'lost' when the program is terminated.

[20] Wikipedia has a useful introduction. See http://en.wikipedia.org/wiki/Object-relational_mapping

[21] Hierarchical Data Format (HDF) technologies are relevant when the data challenges being faced push the limits of what can be addressed by traditional database systems, XML documents, or in-house data formats. See http://www.hdfgroup.org/why_hdf/. *PyTables* is a *Python* API for HDF5. See sonification.com.au/sonipy/dataPersistence.html on the *SoniPy* website for more information.

5.3.2.3 Conceptual modelling and data mapping

In the *SoniPy* model, information mapping is divided into separate cognitive and psychoacoustic stages. The cognitive stage involves the design of 'sound schemas' with semiotics, metaphors and metonyms relating to the task, and aesthetic and compositional aspects relating to genre, culture and palette. Decisions have to be made about functionality, aesthetics, context, learnability, expressiveness, and device characteristics. These decisions are typically drawn from existing knowledge and theories from relevant sciences, arts and design. The consequences of these compounding decisions are difficult to predict empirically: one of the reasons why sonification is currently more of an heuristic art than a science. Nevertheless, as different conceptual models are developed, some based in cognition, others culturally determined, they can be integrated into *SoniPy* using the wrapping techniques outlined.

One example might be the construction of an interface to *TaDa*'s methods; a design approach to sonification that provides a systematic user-centred process to address the multitude of decisions required to design a sonification (Barrass 1996a). *TaDa* starts from a description of a use case scenario, and an analysis of the user's task, and the characteristics of the data. This analysis informs the specification of the information requirements of the sonification. *SoniPy*'s support for the *TaDa* method would be through a python-based GUI that captures a user scenario and provides standard *TaDa* fields for analysis. This GUI, connected using the SDN to a *mySQL* database that contains an increasing number (currently about 200) of stories about everyday listening experiences, analysed using the *TaDa* data-type fields. This database, called *Earbenders*, is a case-based tool for looking up 'sound schemas' at the cognitive design stage (Barrass 1996b). In future, a *Python* interface to *TaDa*'s SonificationDesignPatterns wiki could also be developed as an alternative pattern-language approach for cognitive level design (Barrass 2003).

5.3.2.4 Psychoacoustic modelling

The psychoacoustic modelling stage involves the systematic mapping of information relations in the data to perceptual relations in the sound schema (Barrass 1996c). The concept is that *SoniPy* would provide support for this by allowing interactive reconfiguration of the mapping from information

relations to auditory relations. Changes in this mapping cause the automatic remapping of source information through psychoacoustic algorithms (implemented in the fast array processing tools, *NumPy* and *SciPy*) to produce new sounds and/or rendering controls. For example a change from categorical to ordered information could automatically produce a remapping from a categorical sound (e.g. instrument, object, stream) to an ordered property of a sounding object (e.g. length, excitation, distance).

5.3.2.5 Acoustic modelling

SoniPy provides a user access to many more options than if a music composition or sound synthesis environment was chosen before beginning the development of other aspects of the data sonification framework. For low-level audio work, a *Portaudio* module can be used for audification and as the basis for the development of other such modules should the need arise (Burk and Bencina 2001). *Portaudio* is used via the python wrapper *PyAudio* for the audio streaming in the capital market net returns experiments in the next chapter. *SndObj* ('sound object') is a middle-level toolkit also immediately available in the same manner (Lazzarini 2000). *SndObj*, like many real-time audio applications today, uses the *Portaudio* interface to the audio hardware. *The STK toolkit* also appears to be wrappable (Scavone and Cook 2005), as does the higher-level *RTCmix C++library*,[22] although no attempt has been made by this author to do so. *Csound* has an embedded *Python* API for writing extended Opcodes for a couple of years, and *Csound5*, the latest evolution of this classic computer sound synthesis language, has a python extension wrapper interface. At the time of writing it is undergoing extensive multiple-platform testing.

Whilst some high-level applications such as *Max/MSP* and *PD*, are unlikely to become extension toolkits to *SoniPy*, it is still possible to use them by instantiating them independently and communicating with them using computer music protocol specifications *OSC* (Wright, Freed, and Momeni 2003) and *MIDI*[23]. *SuperCollider3* is a special case because of its inherent modularity: the sound-rendering component, *Scsynth*, can be instantiated as a separate program to the language, *SClang*. Communication tests between this 'external' *SCsynth* over *OSC* using the SoniPy framework as an alternative to

---

22 http://www.rtcmix.org/
23 http://www.midi.org/

Sclang indicate that this is viable. If very low-latency is a requirement, such as may be the case for interactive sonifications, *pySCLang*, originally designed as an alternative to *Sclang* for Windows platforms, enables direct communication with an instantiation of the SC language and its internally embedded sound renderer. Although this seems like a circuitous route, it works and is another example of the robustness of the encapsulating framework approach used in SoniPy.

Non-operating system dependent text–to–speech synthesis will be available through a *PySpeak*, an OSX thread-compliant *Python* API to of *Espeak*[24] [25].

5.3.2.6 Monitoring, feedback and evaluation

Monitoring and Feedback of SDN and SCN can happen via the *Python* interpreter. Having access to an interpreter in order to build a complete sonification by iteratively building on small tests is a powerful aspect of *Python*. Heterogeneous connectivity also allows the consequences of decisions at each stage to be tested on a compound design, thus enabling better understanding and control of non-linear and emergent effects in an overall design.

For cross-platform GUI-building, *wxPython* provides access to *wxWidgets*[26] and *wxGlade*[27] can assist in more-rapid development of GUIs by automatically generating *Python* control code and separating the GUI design and event-handling code. If a relatively consistent interface across all hardware platforms is more desirable, *Tcl* GUI-building tools[28] are available through native *Python* modules.

The inclusion of evaluation modules will make it possible to design different types of empirical experiments, and conduct and analyse the results within a single framework environment, even using website-based surveys if desired. This would be a marked improvement on current public-domain experimental psychology software, as there is currently no such tool that permits separately-threaded audio streaming or soundfile playback. In

---

24 http://espeak.sourceforge.net/

25 The most lauded text–to–speech is probably *Festival* (http://www.cstr.ed.ac.uk/projects/festival/). Initial compatibility caused it to fail the ossification test.

26 http://www.wxwidgets.org/

27 http://wxglade.sourceforge.net/

28 http://www.tcl.tk/

addition, a user-contributed library of experiments for evaluating sonification designs could assist in developing standards for measuring the comparative functionality, aesthetics, learnability, effectiveness, accuracy, expressiveness and other aspects of individual design. While the design and implementation of a complete empirical sonification evaluation environment is beyond the means of many researchers in experimental psychology, the contribution to the design and implementation of such community-supported tools may not be. As already evidenced by such projects as *SciPy*, this approach is currently in other scientific disciplines such as mathematics, physics, chemistry, astronomy, meteorology and genetics, where the design and sharing of experimental tools has long been an integral component of their research endeavours.

5.4 An illustrated metacode example

Example Code 1 is a simple 'high-level' illustration of part of the *SoniPy* framework in action. The first task is to accept a multiplexed meta-tagged data stream from the Australian Stock Exchange (ASX) trading engine. The ASX is medium-sized exchange, on which about 3,500 securities are traded. It generates about 100 MB of trading text data daily, making it impractical to hold enough data in RAM to do all the calculations necessary.

The data is processed into a *MySQL* database using an Object-Relation Mapping paradigm supported by the *sqlobject* module[29]. This abstracts the handling of the dataset, providing an interface between the tables and indices database paradigm and Python's object-orientation. Other modules (such as *mySQLdb*) are available if direct interaction with the database server in *MySQL* code is more appropriate. A list of securities that meet, or are likely to meet, the criteria necessary for a sonification event to initiated, is held in RAM and processed as a multidimensional array using the *NumPy* module. When the criteria are met this data is also used as some of the input parameters to the sound renderer. The *pyspeak* module is invoked to synthesise the name of the security being newly rendered. In this example, the sound is rendered by the *SuperCollider3*'s external synthesis engine, *scsynth,* with which the python *scsynth* module bi-directionally communicates using the *OSC* protocol. This

[29] http://www.sqlobject.org/

permits the use of *SC3*'s *synthdefs* (synthesis definition algorithms) that are capable of responding to the criteria, as established, or as modified in real-time. Other synthesis options, such as the lower-level *pysndobj* or *pyaudio* (the python interface to *portaudio*) are possible, as is the *libsndfile* library for audiofile playback. The code example illustrates how *SoniPy* combines *Python* code, imported 3rd party modules and user-defined scripts. It is meant here to provide a sense of the immediacy and readability of the approach. More detailed code examples can be found in the Appendix 6.

5.5 The *SoniPy* website

A continually evolving website has been established to assist in establishing *SoniPy* as an active public domain project[30]. In order to do this, the site has been structured around five principle activities:

1. A first port-of-call for sonification-related activities using *Python*.
2. A single help location from which module documentation can be accessed. This is a frames portal that functions on similar principles to that of *Sonipy* modules, in general: by wrapping the existing on-line module documentation if it is available. Otherwise, the documentation that accompanies the module's sourcefiles is built in HTML and/or PDF format and served from the SoniPy website instead.
3. Tutorials on the use of the framework to undertake sonification tasks. The experiments outlined in the next chapter will eventually be placed there, for example, and links to independent third-party material such as *figusdevpack* (Fabbri and Chiozo 2008).
4. A storage and download system. This will include a high-level mechanism for automating the meta-bundling, download and installation of modules, component sets and their dependencies, for various hardware and operating systems, using *Python* Egg technology and the *Python* package manager *Easy-Install*[31]. In addition a software license reference for modules aids the collation of modules appropriate in different commercial circumstances.

---

30 http://www.sonification.com.au/sonipy/

31 "Eggs are to *Python* what jars ar to Java." *Python* Eggs are ZIP archives with the file extension .egg. See http://peak.telecommunity.com/DevCenter/PythonEggs.

5. A community-managed forum for both users and developers to facilitate the detailed orderly discussion of aspects SoniPy. At the time of writing, the forum contains a record of the testing and evaluation of many of the modules discussed in this chapter.

```
# Metacode Example; Sonipy Main Loop

# ::::::::::: import the modules needed ::::::::::::::::::::
# for Australian Stock Exchange (ASX) Trading Engine datafeed.

import DataFeeder
import StreamMonitor
            # uses matplotlib for graphical presentation
            # of trading engine stats
import DataToDB
            # includes an import of the sqlobject Module 3,
            # which provides OO Class structure of the
            # Relation-to-Object-Mapping necessary for handling
DBs.
            # For simple DB calls, use mySQLd
      # to simply pass mySQL commands to the server.
import SecurityMaps
      # mapping OO classes for ASX securities.
      # this Module calls the numpy module for
      # fast multi-dimensional array processing.
import PsychoFilters
      # a set of filters for transforming mapped data
      # Used to adjust Security Maps for psychophysical
      # non-linearities, condition enhancement etc
import SoundRenders
      # A set of synthesis engine interfaces and
      # synthesis definitions (instruments) appropriate
      # for this sonification. Includes OSC and MIDI,
      # and the phoneme synthesiser pyspeak.

# ::::::::::: ASX sonification threads :::::::::::::::::::::
ASXFeed=DataFeeder(feedURL='http://localhost')
      # Establish connection to datafeed. The dtafeed
      # URL could be an internet,LAN address or filepath.
      # Uses Python build-in modules for low-level
      # data transfer to RAM buffer.
StreamMonitor.monitorStream (ASXFeed)
      # Monitor data stream.
DataToDB.MultiplexStreamtoDB(ASXFeed, ConnectParams)
            # Start processing ASXFeed into DB. Uses a MySQL DB
            # DB client connection as specified in ConnectParams.
            # Other servers are possible, including remote ones.
ASXalerts=SecurityMaps.Indicator(stocks, UpBollinger(20,9,2))
      # a FIFO of alerts for securities currently trading
            # outside the specified upper Bollinger band. The FIFO
            # is constantly updated whilst DataFeeder is active.
SoundRenderers.BinauralOut='TRUE'
            # Process sound output will for binaural listening.
while (StreamMonitor.StillActive):
    for security in ASXalerts:
        if security not in renderList:
            pyspeak(security)  # announce new security
            SoundRenders.scsynth(security, stock.scsynthdef)
```

Code Example 1. The Sonipy Framework in action. '#' is the Python comment character. The task modelled is to accept data streamed from a stock market-trading engine and use sonification to alert the listener to specific features of the trading activity as given .

5.6 Summary

Sonification research is an interdisciplinary activity and to date, tools for undertaking it have either been discipline-specific, modified to accommodate the interconnections, ad-hoc collections, or stand-alone programs developed for a specific sonification task. Because *SoniPy*'s open architecture design can integrate modules conforming to widely accepted inter-process computation standards (wrappable libraries), in a non-conflicting way, it will be possible for it to grow in the directions its user-community needs it to. Instead of sonification researchers trying to piece together a collection of tools they will hopefully be able to make work together, they will be able to choose from a number of possible *SoniPy* framework modules, based on a best-fit-for-the–task evaluation, and rely on the continuity of the framework to provide inter-module integration.

Because the framework is implemented in a common, 'user-friendly,' scripting language, programming assistance, when needed, will be more readily available than if a specialist application were used, and this may, in term, assist individual endeavour and promote better independent evaluation of the empirical results of other research in sonification. There is currently, for example, a dearth of good public-domain software for experimental psychology that uses sound, and the integration of such a module, or set of modules, into *SoniPy* would be a welcome addition to the field.

It has not been practicable, in the current context, to test all the alternative combinations of modules accessible and useful in undertaking data sonification. The approach taken was to select a number of modules from different domains and test, albeit in a reasonably ad-hoc way, the viability of their combined use to solve a specific problem that would normally be undertaken, often with some awkwardness, using two or more unrelated tools. Once the library encapsulation had been effected, we did not experience a single conflict during this process.

Probably the most important feature of the framework approach outlined in this chapter is that, by using tools largely built by and for large communities of uses with a vested interest in their survival independent of data sonification, sonifiers are less likely to be 'held to ransom' by a reliance on the need for the developers of single software applications to continue to

respond to the ongoing hardware and operating systems development environments in which they operates, and thus risk the isolation and eventual obsolescence that inevitably follows when they do not. *Python*, *NumPy*, *Matplotlib*, *Portaudio* etc will continue to evolve and eventually be replaced by improved alternatives, but in ways that are more adaptive than catastrophic.

While the project is in its infancy, a major concern of the overall design has been the efficiency and effectiveness with which it can be implemented and maintained by a small team of coordinating developers. Whether or not this occurs is dependent on to the vagaries of public forces but there is some evidence that, since the original papers were published (Worrall et al. 2007; Worrall 2008), others have begun to build on this initial work for the *Linux* operating system (Fabbri and Chiozo 2008). It may, however, find its place in the public sphere as a part of a larger project, such as *SciPy*, or the more tightly defined *python(x,y)*[32].

There are some clear, and some not–so–clear distinctions between software *design* and software *engineering* as designer and computer musician Bill Buxton emphasises (2003). Interpretive languages tend to afford individual users the opportunity to do both, but this relies on strong design principles being well implemented, both in the underlying language (*Python* in this instance) as well as extensive framework extensions such as *SoniPy*. By establishing an open source project based on such design principles, it is hoped that the initial work presented here will be taken up by others who, in turn, will contribute an evolving framework that is useful to the wider sonification community; that data sonification software design ‘escapes’ from the engineers to become a more widely accepted part of what it means to ‘do’ sonification than is currently the case.

[32] http://www.pythonxy.com/

THIS PAGE HAS INTENTIONALLY BEEN LEFT BLANK