ipyclam, empowering CLAM with Python

David GARCÍA-GARZÓN
Departament de Tecnologia,
Universitat Pompeu Fabra
Tànger, 122-140
08018 Barcelona
Spain
david.garcia@upf.edu

Xavier SERRA-ROMÁN
Imm Sound, a Dolby Company
Diagonal 177
08018 Barcelona
Spain
Xavi.Serra@dolby.com

Abstract

This paper introduces ipyclam, a new way of manipulating networks in CLAM (C++ Library for Audio and Music) by using the Python language. This extends the power of the framework in many ways. Some of them are exploring and manipulating live processing networks via interactive Python shells, or extending the power of visual prototyping in CLAM by adding complex application logic and user interfaces with PyQt/PySide. The described Python API, ipyclam, by redefining the engine layer, can be reused to control other patching based systems such as JACK, gAlan...

Keywords

Python, CLAM, Qt, patching

1 Introduction

CLAM (C++ Library for Audio and Music)[1] is a free software framework to develop advanced signal processing systems [Amatriain et al., 2007]. Some successful use cases include instruments [Haas, 2001; Mann et al., 2007], voice processing [Sommavilla et al., 2007], audio and music information retrieval [Gómez, 2006; Gouyon, 2005; Amatriain et al., 2005; Ong, 2007], and 3D audio [Arumi et al., 2009; Giulio Cengarle, 2012].

As its name states, CLAM is a C++ framework. General purpose dynamic languages, such as Python, do not mix well with real-time audio programming. Those languages hide aspects that are important to control in real-time programming, for example, memory management and operations that imply system calls that could stall the real-time thread. But real-time restrictions only apply to the processing code. Properly designed audio software separates the real-time code from the rest where those restrictions does not apply: setup, user interface, application logic... CLAM fosters a programming style which clearly localizes real-time code. For the remaining code without real-time restrictions, Python may still have an interesting role to play.

This paper introduces ipyclam, a new way of manipulating CLAM data flow definitions (*networks*) by using the Python language. This extends the power of the framework in many ways. For example, it can be used to build complex networks, like the one shown in Figure 1, that are hard to build by graphical means. Those manipulations could be done interactively, by integrating interactive Python shells like IPython [Pérez and Granger, 2007], into the CLAM patching tool, the NetworkEditor. And last but not least, it extends CLAM graphical prototyping architecture, currently based on graphical design tools that generate fixed data flow and single dialog interfaces. With Python we can add rich application logic and interfaces based on PySide [Bert, 2012] or PyQt [Summerfield, 2007] without raising the difficulty to the point of requiring C++ development.

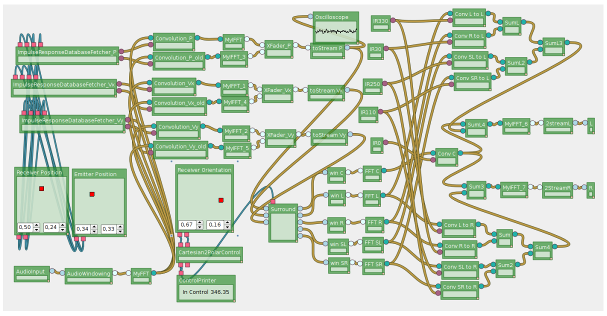


Figure 1: Complex networks are hard to design by pointing and clicking.

The rest of this paper has the following structure: Key concepts of the CLAM framework are introduced in section 2. Section 3 describes the new Python API at user level. Section 4 explains the internal design and how it enables

[1]http://clam-project.org

the reuse of the user API for other patching systems. Section 5 explains how to build PyQt/PySide interfaces that can be related to CLAM networks and how all that leads to a more powerful prototyping architecture. Finally, section 6 evaluates the already reached milestones and the ones that are at reach from now on.

2 CLAM elements

This section will shortly introduce the basic components of the CLAM framework needed to understand this paper. A more insightful description can be found in the referred literature about CLAM.

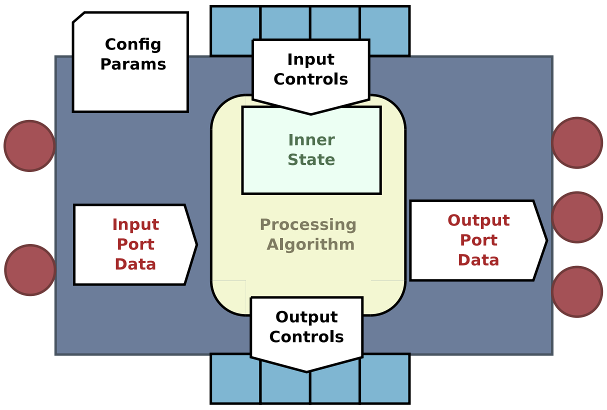


Figure 2: A processing unit

Audio processing is modularized into objects called *processing units* according the CLAM meta-model [Amatriain, 2005]. A processing unit consumes and produces data tokens by its *input and output connectors*. Connectors are called *ports* when data flow is continuous and they are called *controls* when data is sent or received unevenly. Token data can be any C++ type but each connector is bound to a single type. When connecting connectors of different processing units, they must be complementary (input and output), same *kind* (port or control), and same *type* (data token C++ class).

Each processing unit has a set of structured *configuration parameters*. Configuration and connection is done before run-time so that any operation that requires resource allocation can be done outside the real-time thread.

A *network* is a set of interconnected processing units. The network schedules the execution of the units under a given *audio back-end* (PortAudio, JACK, LADSPA, LV2, VST...). Backend data is fed from and to special units inside the network called *sources* and *sinks*. Then the network topology mandates the data-flow scheduling [Arumí, 2009].

UI *binders* are used to relate a CLAM network to a user interface, currently Qt, but not restricted to it. The programmer can establish such relation by defining custom properties on the elements of the user interface (*widgets*). UI binders detect such properties and add any required stuff to bind them to the network. Common examples of UI binders are the ones used to bind user interface for playback control and monitoring, processing unit configuration, data token visualization, and user control sending.

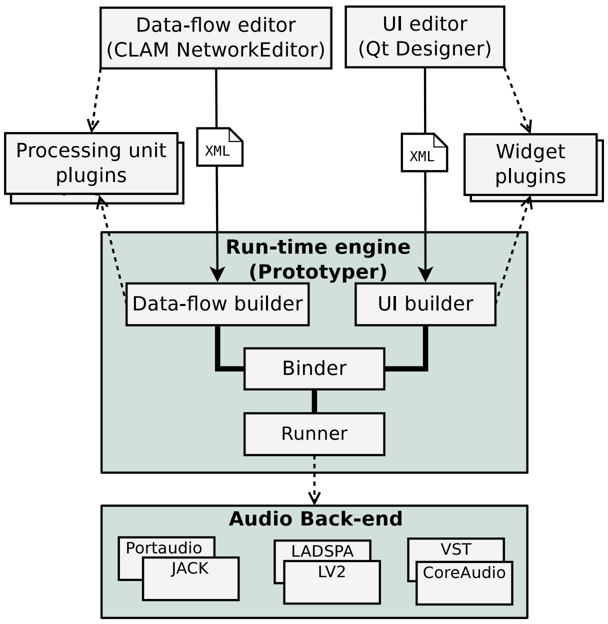


Figure 3: Visual prototyping architecture

All those elements enable the CLAM visual prototyping architecture [Garcia, 2007] illustrated in Figure 3. Both the processing network and the user interface can be designed with graphical tools, CLAM NetworkEditor and Qt Designer respectively. Both can be stored as XML, loaded later in run-time, and related by applying the binders. A tool called Prototyper does that by taking the XML files by command line.

Most elements in this architecture (processing units, token data type handlers, back-ends, UI binders, widgets...) can be extended via plugins. Most of those extendible objects are available through abstract interfaces and factories.

3 ipyclam user API

3.1 Goals

ipyclam's main goal is providing the API to be able to build and explore a CLAM network. Defining the processing code inside processing units is reserved to C++ code to fit real-time constraints.

An explicit choice has been taken on not designing Python API as a direct map of the C++ CLAM API, but to make it conveniently Pythonic. Direct C++ library mapping often leads to a badly designed Python interface. Design decisions taken in C++ API just because of C++ idiosyncrasy, may get pointlessly replicated in Python, and opportunities of using Python features such as attributes, iterators, generators or dynamic interface creation, may get lost.

Another choice is to provide a powerful tab completion for interactive Python shell. It is not just about discovering the static API but taking a step further and discovering the runtime structure of the objects via tab completion. Because of that, such structure should be available as completable attributes.

Convenient ways of expressing things are favored but whenever those convenient ways are not expressive enough to express everything, instead of discarding the convenient one, we add the less convenient one as alternative.

For example, processing units are accessible as network attributes with their names. Also processing ports, controls and configuration parameters are accessible as processing attributes as well. That interface is compact and convenient when doing tab completion.

```
net.Sink.Audio
```

But this is not a general solution. Many names are not valid identifiers. Subscript accessors are provided to solve those cases:

```
net["A processing"]["1"]
```

Still, you may find two subelements of different kind with the same name, or with a name that matches an actual attribute or method of the object. For those cases, it is useful to provide scoping attributes. So the syntax that will always work would be:

```
net.processings["Sink"].inports["Audio"]
```

But this is more verbose than the first proposal. It is a design choice when this kind of situation appears, to provide both the convenient and the complete options so we get the best of them.

3.2 An example

This is a minimal example that creates a 3 channel cable, that just copies the input to the output, and plays it under JACK:

```
from ipyclam import Network, time
net = Network()
# creating units
net.source = "AudioSource"
net.sink = net.types.AudioSink
# configuring
net.source.NSources = 3
net.sink.NSinks = 3
# connecting
net.source > net.sink
# Playing as JACK client for 1 minute
net.backend = "JACK"
net.play()
time.sleep(60)
net.stop()
```

3.3 Creating processing units

Notice that the first processing in the example, *source*, is created by assigning a string, the processing type name, to a new attribute with the name of the processing. The second one, *sink*, is created instead by using the 'net.types' object. Such object is convenient for interactive use to discover the available types by tab completion.

```
>> net.types.Audio [tab]
AudioSink, AudioMixer, AudioSource
...
```

If the unit name is not a proper Python identifier, the subscript syntax can be used as well:

```
net["My Sink"] = net.types.AudioSink
```

3.4 Configuring

Configuration parameters can be accessed directly as direct attribute or subscript of the processing unit. They can be accessed as well inside the scoping attribute *config* to avoid conflicts with other processing subelements or common attributes and methods.

```
net.source.config.NSources = 3
```

Every time a parameter is set, the object is reconfigured, but reconfiguration may be an expensive process. To address that issue reconfiguration may be held while setting a set of parameters for a given unit using the *with* statement:

```
with net.mymodule.config as c :
  c.AParameter = "A Value"
  c.AnotherParameter = 23.2
```

Configuration parameters are typed, type checking is done on assignment rising *TypeError* if the type is not the proper one. In CLAM, configuration parameters can be instantiated or not. In Python uninstantiated state is represented by the *None* value.

Some configurations are structured using parameters that are configurations themselves. Such sub-configurations can be accessed as natural by accessing successive attributes.

```
net.mymodule.SubConfig.Param1 = 4
```

3.5 Connecting

The example uses the greater-than operator to establish the connection. Both sides of the operator refer to the processing units, but indeed what gets connected are the connectors. So this is a short-cut for connecting each port pair-wise:

```
net.source["1"] > net.sink["1"]
net.source["2"] > net.sink["2"]
net.source["3"] > net.sink["3"]
```

Or, generally, by using the iterators of *inports* and *outports* attributes:

```
for inport, outport in zip(
      net.source.outports,
      net.sink.inports,
    ) :
    inport > outport
```

Similar iteration can be done with *incontrols* and *outcontrols* processing unit attributes. They can be used as well with Python slices. For example, if we want to reverse the channels:

```
net.source > net.sink.inports[::-1]
```

Or first and third to the first two:

```
net.source.outports[::2] > \
  net.sink.inports[:2]
```

3.6 Playback control

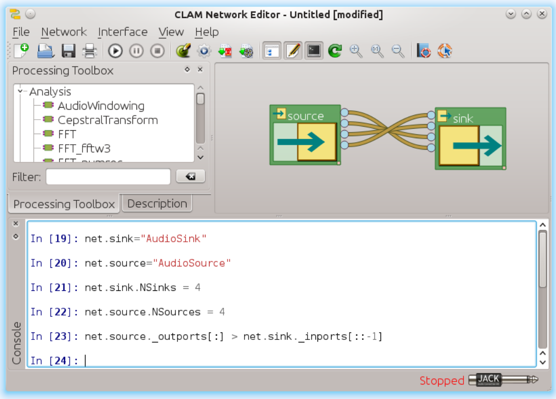
The audio back-end can be set by assigning the *backend* special network attribute. For example, if we wanted to use the PortAudio audio backend we could use:

```
net.backend = "PortAudio"
```

The network has several methods, *pause()*, *play()* and *stop()*, to control the playback, and several methods, *isPlaying()*, *isPaused()* and *isStopped()*, to query the playback status.

3.7 Serialization

Networks can be loaded from XML files generated by NetworkEditor.



```
In [19]: net.sink="AudioSink"

In [20]: net.source="AudioSource"

In [21]: net.sink.NSinks = 4

In [22]: net.source.NSources = 4

In [23]: net.source._outports[:] > net.sink._inports[::-1]

In [24]: |
```

Figure 4: IPython console integrated in the NetworkEditor interface

```
net = Network()
net.load("mynetwork.clamnetwork")
net.save("mynetwork-copy.clamnetwork")
```

Indeed you can get the XML string for the current network using *xml()*.

```
print net.xml()
```

Although XML is somehow readable, in fact, we found that Python code is even more readable than XML. An ipyclam network is able to generate code to reconstruct itself.

```
>> print net.code("mynet")
mynet = Network()
mynet.sink = "AudioSink"
nynet.sink.NSinks = 3
...
```

This feature is quite powerful. Given a static network stored as XML, it can be converted to Python code and as Python code it can be parametrized or turned into a more smart program.

This also opens the door to the use of Python code as serialization format instead of XML. Indeed Python code using ipyclam API is more compact and readable than XML. Despite that, deprecating XML is not yet an option as it is not save to use Python interpreter as parser. A Python interpreter will allow to execute more than just network definitions.

4 Implementation

This section gives a slight overview on how ipyclam API has been internally implemented and how this design allows extending the use of the API to control other patching systems.

ipyclam is designed in two layers as shown in Figure 5. The user layer is the one that provides

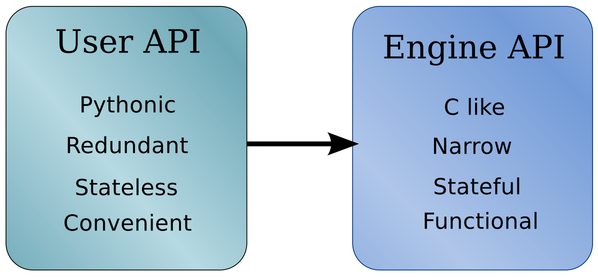


Figure 5: Two layers architecture

the API explained on previous sections, with all the sugar for the many redundant and pythonic ways of expressing the same operation.

But that layer is stateless. In order to perform the actual operations it relies on a engine layer which holds the actual state, in this case, the C++ CLAM Network object. Those many ways of performing a given operation at the user API converge in a single entry point at the engine layer resulting in a narrower API at that level.

This design in two layers strengthens the reliability of the implementation. The user API can be developed ignoring all the complexities of the adapters to the C++ CLAM code by providing a mock-up engine in pure Python. A narrow engine API reduces the number of operations to test for the engine and centralizes the state checks for the front-end testing. A stateless front-end avoids errors on the bookkeeping of duplicated information.

Another positive side effect of this design is that this narrow engine API can be reimplemented to address any other patch like systems, such as JACK, Patchage, gAlan... As result all the rich ipyclam API interface can be reused for those systems. Other patching programs can integrate the Qt console like the one that now NetworkEditor has and is shown in Figure 4.

5 Prototyping user interfaces

CLAM visual prototyping architecture, explained in section 2, provided a way to build a simple audio application by joining two parts designed visually: a CLAM network and a Qt Designer interface. Although that architecture generated decent applications, it has a clear ceiling of what you can build. Applications are limited to simple application logic, a single dialog and a fixed processing data-flow. If anyone wants to go beyond that, C++ programming skills are required, so the learning threshold goes up and the development work-flow gets harder and slower [Garcia, 2007].

An intermediate solution is to introduce Python as programming language for the user interface and application logic. Python is easier to learn and has a faster development work-flow. This section explains some features of ipyclam that facilitate building such applications and shows some examples that illustrate the scope of what you can do.

5.1 PyQt4 and PySide

Two Python bindings are available for Qt: PyQt4[2] and PySide[3]. Each one uses a different binding generator technology: PyQt4 uses SIP while PySide uses Shiboken. The resulting Python APIs are mostly identical, so writing Python code that works for either is not hard. ipyclam supports both. In the following examples, PyQt4 is used but using PySide is just a matter of changing the `import` lines.

5.2 A Python based Prototyper

The following Python code provides a simplified version of Prototyper.

```
import ipyclam, sys
from PyQt4 import QtGui
import ipyclam.ui.PyQt4 as ui
# network setup
net = Network()
net.backend = "JACK"
net.load(sys.argv[1])
# ui setup
app = QtGui.QApplication([])
w = ui.loadUi(sys.argv[2])
net.bindUi(w)
# run
w.show()
net.play()
app.exec_()
net.stop()
```

The interesting bits are the `loadUi` function from the `ipyclam.ui.PyQt4` module and the `bindUi` method of the network. The `loadUi` function is a helper that instantiates a Qt Designer file. The `bindUi` method applies all the available binders to the user interface. Possible bindings are searched recursively so you can use it with a full interface as well as a single widget.

This snippet has the same restrictions as Prototyper: It is general but it is limited to a single processing data flow and a single interface with no application logic.

[2]http://www.riverbankcomputing.com/software/pyqt
[3]http://www.pyside.org

The good news is that now we can change that code to modify the network with the ipyclam API exposed on previous versions and modify the interface with regular PyQt4/PySide API.

5.3 Building interfaces from scratch

A counterexample would be building the processing network and the interface without XML files, that is, using ipyclam and PyQt4/PySide APIs. A problem with this approach is that some useful audio widgets provided by CLAM as Qt plugins have no specific Python wrappers. Providing such wrappers would imply to generate them for SIP and Shiboken for each specific widget class in the plugin. Instead, ipyclam provides a helper method to access the Qt widget factory, which creates the widgets from the class name string. Factory created widgets are handled by the generic QWidget interface, which includes composing them and accessing their properties.

The following example implements an oscilloscope, by binding a CLAM Oscilloscope widget with an AudioSource.

```
import ipyclam, sys
from PyQt4 import QtGui
import ipyclam.ui.PyQt4 as ui
# network setup
net = Network()
net.backend = "JACK"
net.source = net.types.AudioSource
# ui setup
app = QtGui.QApplication([])
w = ui.createWidget("Oscilloscope")
w.setProperty("lineColor", "red")
w.setProperty(
  "clamOutPort", "source.1")
net.bindUi(w)
# run
w.show()
net.play()
app.exec_()
net.stop()
```

This example accesses specific behaviour of the Oscilloscope, the *lineColor*, by using the generic property interface. The same method is used to set the binding property *clamOutPort* that in a visually designed prototype should have been defined with Qt Designer.

5.4 Hybrid approaches

Any combined approach is feasible. Figure 6 shows an example that comes with ipyclam that combines a Qt Designer file with a coded interface. Indeed, this example has some application logic not available with simple visual prototyping. Notice that the combo box is filled with information, the names of the processing units, taken from the network with the ipyclam API. The configure button, instead of being a bound widget, activates a function that takes the currently selected processing unit, and launches a configuration dialog bound to the given processing configuration.

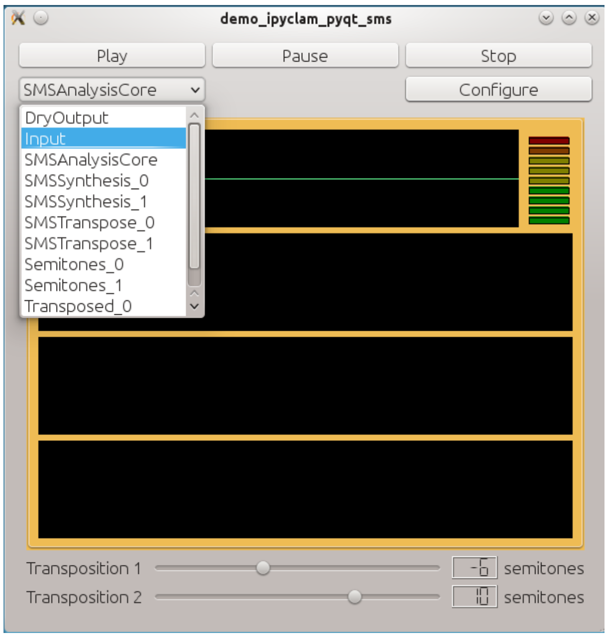


Figure 6: Extending with Python an existing visual prototype that uses Spectral Modeling Synthesis for a two voices transposition. The extension provides detailed configuration of every unit.

6 Conclusions

The API presented in this paper offers a new way of developing real-time audio applications by combining the power and flexibility of CLAM, Qt and Python. The API has been designed with a strong stress on convenience and expressiveness which results in very readable and compact code.

An interactive Python console has been integrated with the graphical patching tool. This enables the user to build complex networks by interactive programming, and having visual feedback of the results. This work can be easily extended to other patching systems just by implementing a narrow API.

Indeed, a promising engine to implement in the future is one relying on JACK because CLAM users are likely to be interested in controlling JACK application interconnections from the console, just as they control inner units.

Another work to be done is providing some useful examples built with ipyclam that give potential users a clear idea of the horizons of the platform. They also will help to mature the API highlighting any unpolished edges left.

Right now, the platform excludes Python for processing tasks. But Python has a nice collection of numerical libraries based on the numpy package [Ascher et al., 1999]. They could be used for processing algorithms for off-line processing or situations where lesser real-time conditions are required. Two approaches are being considered. One is being able to implement processing units in Python. The other is a Python audio back-end where Python code feeds the network with numpy arrays as audio input and output.

7 Acknowledgements

We would like to thank Pau Arumí, Natanael Olaiz and Eduard Aylon for helping us to define and test the API. Part of this work has been done using technical resources gently offered by Fundació Barcelona Media and ImmSound.

References

Xavier Amatriain, Jordi Massaguer, David Garcia, and Ivan Mosquera. 2005. The clam annotator a cross-platform audio descriptors editing tool. 1

Xavier Amatriain, Pau Arumi, and David Garcia. 2007. A framework for efficient and rapid development of cross-platform audio applications. *ACM Multimedia Systems Journal*. 1

Xavier Amatriain. 2005. *An Object-Oriented Metamodel for Digital Signal Processing with a focus on Audio and Music*. Ph.D. thesis, Universitat Pompeu Fabra. 2

Pau Arumi, Natanael Olaiz, and Toni Mateos. 2009. Remastering of movie soundtracks into immersive 3D audio. In *Proceedings of Blender Conference 2009*. 1

Pau Arumí. 2009. *Real-time Multimedia Computing on Off-the-Shelf Operating Systems: From Timeliness Dataflow Models to Pattern Languages*. Ph.D. thesis, Universitat Pompeu Fabra. Master Thesis. 2

David Ascher, Paul F. Dubois, Konrad Hinsen, James Hugunin, and Travis Oliphant, 1999. *Numerical Python*. Lawrence Livermore National Laboratory, Livermore, CA, ucrl-ma-128569 edition. 7

A.C. Bert. 2012. *Pyside*. Chromo Publishing. 1

David Garcia. 2007. Visual Prototyping of Audio Applications. Master's thesis, Universitat Pompeu Fabra. 2, 5

Giulio Cengarle. 2012. *3D audio technologies: applications to sound capture, post-production and listener perception*. Ph.D. thesis, Universitat Pompeu Fabra. 1

Fabien Gouyon. 2005. *A computational approach to rhythm description — Audio features for the computation of rhythm periodicity functions and their use in tempo induction and music content processing*. Ph.D. thesis, Universitat Pompeu Fabra. 1

Emilia Gómez. 2006. *Tonal Description of Music Audio Signals*. Ph.D. thesis, Universitat Pompeu Fabra. 1

Joachim Haas. 2001. Salto - a spectral domain saxophone synthesizer. In *Proceedings of Mosart Conference 2001*. 1

Steve Mann, Ryan Janzen, and James Meier. 2007. The electric hydraulophone: A hyperacoustic instrument with acoustic feedbacks. In *Proceedings of the 2007 International Computer Music Conference (ICMC2007)*, pages 27–31. 1

Bee Suan Ong. 2007. *Structural Analysis and Segmentation of Music Signals*. Ph.D. thesis, University Pompeu Fabra, Barcelona, Spain, February. 1

Fernando Pérez and Brian E. Granger. 2007. IPython: a System for Interactive Scientific Computing. *Comput. Sci. Eng.*, 9(3):21–29, May. 1

Giacomo Sommavilla, Carlo Drioli, Piero Cosi, and Giulio Paci. 2007. SMS-FESTIVAL: a New TTS Framework. In *Models and analysis of vocal emissions for biomedical applications: 5th International workshop*, pages 89–92. Firenze University Press, December 13-15. 1

Mark Summerfield. 2007. *Rapid gui programming with python and qt: the definitive guide to pyqt programming*. Prentice Hall Press, Upper Saddle River, NJ, USA, first edition. 1