Abstract
This paper discusses the use of Python for developing audio signal processing applications. Overviews of Python language, NumPy, SciPy and Matplotlib are given, which together form a powerful platform for scientific computing. We then show how SciPy was used to create two audio programming libraries, and describe ways that Python can be integrated with the SndObj library and Pure Data, two existing environments for music composition and signal processing.

Keywords
Audio, Music, Signal Processing, Python, Programming

1 Introduction
There are many problems that are common to a wide variety of applications in the field of audio signal processing. Examples include procedures such as loading sound files or communicating between audio processes and sound cards, as well as digital signal processing (DSP) tasks such as filtering and Fourier analysis [Allen and Rabiner, 1977]. It often makes sense to rely on existing code libraries and frameworks to perform these tasks. This is particularly true in the case of building prototypes, a practise common to both consumer application developers and scientific researchers, as these code libraries allows the developer to focus on the novel aspects of their work.

Audio signal processing libraries are available for general purpose programming languages such as the GNU Scientific Library (GSL) for C/C++ [Galassi et al., 2009], which provides a comprehensive array of signal processing tools. However, it generally takes a lot more time to develop applications or prototypes in C/C++ than in a more lightweight scripting language. This is one of the reasons for the popularity of tools such as MATLAB [MathWorks, 2010], which allow the developer to easily manipulate matrices of numerical data, and includes implementations of many standard signal processing techniques. The major downside to MATLAB is that it is not free and not open source, which is a considerable problem for researchers who want to share code and collaborate. GNU Octave [Eaton, 2002] is an open source alternative to MATLAB. It is an interpreted language with a syntax that is very similar to MATLAB, and it is possible to write scripts that will run on both systems. However, with both MATLAB and Octave this increase in short-term productivity comes at a cost. For anything other than very basic tasks, tools such as integrated development environments (IDEs), debuggers and profilers are certainly a useful resource if not a requirement. All of these tools exist in some form for MATLAB/Octave, but users must invest a considerable amount of time in learning to use a programming language and a set of development tools that have a relatively limited application domain when compared with general purpose programming languages. It is also generally more difficult to integrate MATLAB/Octave programs with compositional tools such as Csound [Vercoe et al., 2011] or Pure Data [Puckette, 1996], or with other technologies such as web frameworks, cloud computing platforms and mobile applications, all of which are becoming increasingly important in the music industry.

For developing and prototyping audio signal processing applications, it would therefore be advantageous to combine the power and flexibility of a widely adopted, open source, general purpose programming language with the quick development process that is possible when using interpreted languages that are focused on signal processing applications. Python [van Rossum and Drake, 2006], when used in conjunction with the extension modules NumPy [Oliphant, 2006], SciPy [Jones et al., 2001] and Matplotlib [Hunter, 2007] has all of these characteristics.
Section 2 provides a brief overview of the Python programming language. In Section 3 we discuss NumPy, SciPy and Matplotlib, which add a rich set of scientific computing functions to the Python language. Section 4 describes two libraries created by the authors that rely on SciPy, Section 5 shows how these Python programs can be integrated with other software tools for music composition, with final conclusions given in Section 6.

2 Python

Python is an open source programming language that runs on many platforms including Linux, Mac OS X and Windows. It is widely used and actively developed, has a vast array of code libraries and development tools, and integrates well with many other programming languages, frameworks and musical applications. Some notable features of the language include:

- It is a mature language and allows for programming in several different paradigms including imperative, object-orientated and functional styles.
- The clean syntax puts an emphasis on producing well structured and readable code. Python source code has often been compared to executable pseudocode.
- Python provides an interactive interpreter, which allows for rapid code development, prototyping and live experimentation.
- The ability to extend Python with modules written in C/C++ means that functionality can be quickly prototyped and then optimised later.
- Python can be embedded into existing applications.
- Documentation can be generated automatically from the comments and source code.
- Python bindings exist for cross-platform GUI toolkits such as Qt [Nokia, 2011].
- The large number of high-quality library modules means that you can quickly build sophisticated programs.

A complete guide to the language, including a comprehensive tutorial is available online at http://python.org.

3 Python for Scientific Computing

Section 3.1 provides an overview of three packages that are widely used for performing efficient numerical calculations and data visualisation using Python. Example programs that make use of these packages are given in Section 3.2.

3.1 NumPy, SciPy and Matplotlib

Python’s scientific computing prowess comes largely from the combination of three related extension modules: NumPy, SciPy and Matplotlib. NumPy [Oliphant, 2006] adds a homogenous, multidimensional array object to Python. It also provides functions that perform efficient calculations based on array data. NumPy is written in C, and can be extended easily via its own C-API. As many existing scientific computing libraries are written in Fortran, NumPy comes with a tool called f2py which can parse Fortran files and create a Python extension module that contains all the subroutines and functions in those files as callable Python methods.

SciPy builds on top of NumPy, providing modules that are dedicated to common issues in scientific computing, and so it can be compared to MATLAB toolboxes. The SciPy modules are written in a mixture of pure Python, C and Fortran, and are designed to operate efficiently on NumPy arrays. A complete list of SciPy modules is available online at http://docs.scipy.org, but examples include:

File input/output (scipy.io): Provides functions for reading and writing files in many different data formats, including .wav, .csv and matlab data files (.mat).

Fourier transforms (scipy.fftpack): Contains implementations of 1-D and 2-D fast Fourier transforms, as well as Hilbert and inverse Hilbert transforms.

Signal processing (scipy.signal): Provides implementations of many useful signal processing techniques, such as waveform generation, FIR and IIR filtering and multi-dimensional convolution.

Interpolation (scipy.interpolate): Consists of linear interpolation functions and cubic splines in several dimensions.

Matplotlib is a library of 2-dimensional plotting functions that provides the ability to quickly visualise data from NumPy arrays, and produce publication-ready figures in a variety of formats. It can be used interactively from the Python command prompt, providing similar functionality to MATLAB or GNU Plot [Williams et al., 2011]. It can also be used in Python scripts, web applications servers or in combination with several GUI toolkits.
3.2 SciPy Examples

Listing 1 shows how SciPy can be used to read in the samples from a flute recording stored in a file called *flute.wav*, and then plot them using Matplotlib. The call to the `read` function on line 5 returns a tuple containing the sampling rate of the audio file as the first entry and the audio samples as the second entry. The samples are stored in a variable called `audio`, with the first 1024 samples being plotted in line 8. In lines 10, 11 and 13 the axis labels and the plot title are set, and finally the plot is displayed in line 15. The image produced by Listing 1 is shown in Figure 1.

```python
from scipy.io.wavfile import read
import matplotlib.pyplot as plt

# read audio samples
input_data = read("flute.wav")
audio = input_data[1]
# plot the first 1024 samples
plt.plot(audio[0:1024])
# label the axes
plt.ylabel("Amplitude")
plt.xlabel("Time (samples)")
# set the title
plt.title("Flute Sample")
# display the plot
plt.show()
```

Listing 1: Plotting Audio Files

![Flute Sample](image)

Figure 1: Plot of audio samples, generated by the code given in Listing 1.

In Listing 2, SciPy is used to perform a Fast Fourier Transform (FFT) on a windowed frame of audio samples then plot the resulting magnitude spectrum. In line 11, the SciPy `hann` function is used to compute a 1024 point Hanning window, which is then applied to the first 1024 flute samples in line 12. The FFT is computed in line 14, with the complex coefficients converted into polar form and the magnitude values stored in the variable `mags`. The magnitude values are converted from a linear to a decibel scale in line 16, then normalised to have a maximum value of 0 dB in line 18. In lines 20-26 the magnitude values are plotted and displayed. The resulting image is shown in Figure 2.

```python
import scipy
from scipy.io.wavfile import read
from scipy.signal import hann
from scipy.fftpack import rfft
import matplotlib.pyplot as plt

# read audio samples
input_data = read("flute.wav")
audio = input_data[1]
# apply a Hanning window
window = hann(1024)
audio = audio[0:1024] * window
# fft
mags = abs(rfft(audio))
# convert to dB
mags = 20 * scipy.log10(mags)
# normalise to 0 dB max
mags -= max(mags)
# plot
plt.plot(mags)
# label the axes
plt.ylabel("Magnitude (dB)")
plt.xlabel("Frequency Bin")
# set the title
plt.title("Flute Spectrum")
plt.show()
```

Listing 2: Plotting a magnitude spectrum

4 Audio Signal Processing With Python

This section gives an overview of how SciPy is used in two software libraries that were created by the authors. Section 4.1 gives an overview of Simpl [Glover et al., 2009], while Section 4.2 introduces Modal, our new library for musical note onset detection.

4.1 Simpl

Simpl \(^1\) is an open source library for sinusoidal modelling [Amatriain et al., 2002] written in C/C++ and Python. The aim of this project is

\(^1\)Available at http://simplsound.sourceforge.net
to tie together many of the existing sinusoidal modelling implementations into a single unified system with a consistent API, as well as provide implementations of some recently published sinusoidal modelling algorithms. Simpl is primarily intended as a tool for other researchers in the field, allowing them to easily combine, compare and contrast many of the published analysis/synthesis algorithms.

Simpl breaks the sinusoidal modelling process down into three distinct steps: peak detection, partial tracking and sound synthesis. The supported sinusoidal modelling implementations have a Python module associated with every step which returns data in the same format, irrespective of its underlying implementation. This allows analysis/synthesis networks to be created in which the algorithm that is used for a particular step can be changed without effecting the rest of the network. Each object has a method for real-time interaction as well as non-real-time or batch mode processing, as long as these modes are supported by the underlying algorithm.

All audio in Simpl is stored in NumPy arrays. This means that SciPy functions can be used for basic tasks such as reading and writing audio files, as well as more complex procedures such as performing additional processing, analysis or visualisation of the data. Audio samples are passed into a `PeakDetection` object for analysis, with detected peaks being returned as NumPy arrays that are used to build a list of `Peak` objects. Peaks are then passed to `PartialTracking` objects, which return partials that can be transferred to `Synthesis` objects to create a NumPy array of synthesised audio samples. Simpl also includes a module with plotting functions that use Matplotlib to plot analysis data from the peak detection and partial tracking analysis phases.

An example Python program that uses Simpl is given in Listing 3. Lines 6-8 read in the first 4096 sample values of a recorded flute note. As the default hop size is 512 samples, this will produce 8 frames of analysis data. In line 10 a `SndObjPeakDetection` object is created, which detects sinusoidal peaks in each frame of audio using the algorithm from The SndObj Library [Lazzarini, 2001]. The maximum number of detected peaks per frame is limited to 20 in line 11, before the peaks are detected and returned in line 12. In line 15 a `MQPartialTracking` object is created, which links previously detected sinusoidal peaks together to form partials, using the McAulay-Quatieri algorithm [McAulay and Quatieri, 1986]. The maximum number of partials is limited to 20 in line 16 and the partials are detected and returned in line 17. Lines 18-25 plot the partials, set the figure title, label the axes and display the final plot as shown in Figure 3.

```
import simpl
import matplotlib.pyplot as plt
from scipy.io.wavfile import read

# read audio samples
audio = read("flute.wav")

# take just the first few frames
audio = audio[0:4096]

# Peak detection with SndObj
pd = simpl.SndObjPeakDetection()
pd.max_peaks = 20
pks = pd.find_peaks(audio)

# Partial Tracking with
# the McAulay-Quatieri algorithm
pt = simpl.MQPartialTracking()
pt.max_partials = 20
partls = pt.find_partials(pks)

# plot the detected partials
simpl.plot.plot_partials(partls)
set title and label axes
plt.title("Flute Partials")
plt.ylabel("Frequency (Hz)")
plt.xlabel("Frame Number")
plt.show()
```

Listing 3: A Simpl example
4.2 Modal

Modal is a new open source library for musical onset detection, written in C++ and Python and released under the terms of the GNU General Public License (GPL). Modal consists of two main components: a code library and a database of audio samples. The code library includes implementations of three widely used onset detection algorithms from the literature and four novel onset detection systems created by the authors. The onset detection systems can work in a real-time streaming situation as well as in non-real-time. For more information on onset detection in general, a good overview is given in Bello et al. (2005).

The sample database contains a collection of audio samples that have creative commons licensing allowing for free use and redistribution, together with hand-annotated onset locations for each sample. It also includes an application that allows for the labelling of onset locations in audio files, which can then be added to the database. To the best of our knowledge, this is the only freely distributable database of audio samples together with their onset locations that is currently available. The Sound Onset Labellizer [Leveau et al., 2004] is a similar reference collection, but was not available at the time of publication. The sample set used by the Sound Onset Labellizer also makes use of files from the RWC database [Goto et al., 2002], which although publicly available is not free and does not allow free redistribution.

Modal makes extensive use of SciPy, with NumPy arrays being used to contain audio samples and analysis data from multiple stages of the onset detection process including computed onset detection functions, peak picking thresholds and the detected onset locations, while Matplotlib is used to plot the analysis results. All of the onset detection algorithms were written in Python and make use of SciPy’s signal processing modules. The most computationally expensive part of the onset detection process is the calculation of the onset detection functions, so Modal also includes C++ implementations of all onset detection function modules. These are made into Python extension modules using SWIG [Beazley, 2003]. As SWIG extension modules can manipulate NumPy arrays, the C++ implementations can be seamlessly interchanged with their pure Python counterparts. This allows Python to be used in areas that it excels in such as rapid prototyping and in “glueing” related components together, while languages such as C and C++ can be used later in the development cycle to optimise specific modules if necessary.

Listing 4 gives an example that uses Modal, with the resulting plot shown in Figure 4. In line 12 an audio file consisting of a sequence of percussive notes is read in, with the sample values being converted to floating-point values between -1 and 1 in line 14. The onset detection process in Modal consists of two steps, creating a detection function from the source audio and then finding onsets, which are peaks in this detection function that are above a given threshold value. In line 16 a ComplexODF object is created, which calculates a detection function based on the complex domain phase and energy approach described by Bello et al. (2004). This detection function is computed and saved in line 17. Line 19 creates an OnsetDetection object which finds peaks in the detection function that are above an adaptive median threshold [Brossier et al., 2004]. The onset locations are calculated and saved on lines 21-22. Lines 24-42 plot the results. The figure is divided into 2 subplots, the first (upper) plot shows the original audio file (dark grey) with the detected onset locations (vertical red dashed lines). The second (lower) plot shows the detection function (dark grey) and the adaptive threshold value (green).

```
1 from modal.onsetdetection 
2 import OnsetDetection
3 from modal.detectionfunctions 
```
import ComplexODF
from modal.ui.plot import (plot_detection_function,
plot_onsets)
import matplotlib.pyplot as plt
from scipy.io.wavfile import read

# read audio file
audio = read("drums.wav")[1]
# values between -1 and 1
audio = audio / 32768.0
# create detection function
cdf = ComplexODF()
odf = cdf.process(audio)
# create onset detection object
od = OnsetDetection()
hop_size = cdf.get_hop_size()
onsets = od.find_onsets(odf) * \n    hop_size
# plot onset detection results
plt.subplot(2,1,1)
plt.title("Audio And Detected "
    "Onsets")
plt.ylabel("Sample Value")
plt.xlabel("Sample Number")
plt.plot(audio, "0.4")
plot_onsets(onsets)
plt.subplot(2,1,2)
plt.title("Detection Function "
    "And Threshold")
plt.ylabel("Detection Function "
    "Value")
plt.xlabel("Sample Number")
plot_detection_function(odf, hop_size)
thresh = od.threshold
plot_detection_function(thresh, hop_size, "green")
plt.show()

Listing 4: Modal example

5 Integration With Other Music Applications

This section provides examples of SciPy integration with two established tools for sound design and composition. Section 5.1 shows SciPy integration with The SndObj Library, with Section 5.2 providing an example of using SciPy in conjunction with Pure Data.

5.1 The SndObj Library

The most recent version of The SndObj Library comes with support for passing NumPy arrays to and from objects in the library, allowing data to be easily exchanged between SndObj and SciPy audio processing functions. An example of this is shown in Listing 5. An audio file is loaded in line 8, then the scipy.signal module is used to low-pass filter it in lines 10-15. The filter cutoff frequency is given as 0.02, with 1.0 being the Nyquist frequency. A SndObj called obj is created in line 21 that will hold frames of the output audio signal. In lines 24 and 25, a SndRTIO object is created and set to write the contents of obj to the default sound output. Finally in lines 29-33, each frame of audio is taken, copied into obj and then written to the output.

from sndobj import \\nSndObj, SndRTIO, SND_OUTPUT
import scipy as sp
from scipy.signal import firwin
from scipy.io.wavfile import read

# read audio file
audio = read("drums.wav")[1]
# use SciPy to low pass filter
order = 101
cutoff = 0.02
filter = firwin(order, cutoff)
audio = sp.convolve(audio,
    filter,
    "same")
# convert to 32-bit floats
audio = sp.asarray(audio,
    sp.float32)

# create a SndObj that will hold
# frames of output audio
obj = SndObj()
# create a SndObj that will
# output to the sound card
outp = SndRTIO(1, SND_OUTPUT)
outp.SetOutput(1, obj)
# get the default frame size
f_size = outp.GetVectorSize()
# output each frame
i = 0
while i < len(audio):
    f = audio[i:i+f_size]
    p = m.process(f)
    p = sp.fromstring(p, int16)
    out = sp.hstack((out, p))
    i += f_size
# close the patch
pd.libpd_close_patch(patch)
# write the audio file to disk
write("out.wav", 44100, out)

Listing 5: The SndObj Library and SciPy

5.2 Pure Data
The recently released libpd \(^3\) allows Pure Data to be embedded as a DSP library, and comes with a SWIG wrapper enabling it to be loaded as a Python extension module. Listing 6 shows how SciPy can be used in conjunction with libpd to process an audio file and save the result to disk. In lines 7-13 a PdManager object is created, that initialises libpd to work with a single channel of audio at a sampling rate of 44.1 KHz. A Pure Data patch is opened in lines 14-16, followed by an audio file being loaded in line 20. In lines 22-29, successive audio frames are processed using the signal chain from the Pure Data patch, with the resulting data converted into an array of integer values and appended to the out array. Finally, the patch is closed in line 31 and the processed audio is written to disk in line 33.

```python
import scipy as sp
from scipy import int16
from scipy.io.wavfile import read, write
import pylibpd as pd
num_chans = 1
```

\(^3\)Available at http://gitorious.org/pdlib/libpd

6 Conclusions
This paper highlighted just a few of the many features that make Python an excellent choice for developing audio signal processing applications. A clean, readable syntax combined with an extensive collection of libraries and an unrestricted open source license make Python particularly well suited to rapid prototyping and make it an invaluable tool for audio researchers. This was exemplified in the discussion of two open source signal processing libraries created by the authors that both make use of Python and SciPy: Simpl and Modal. Python is easy to extend and integrates well with other programming languages and environments, as demonstrated by the ability to use Python and SciPy in conjunction with established tools for audio signal processing such as The SndObj Library and Pure Data.

7 Acknowledgements
The authors would like to acknowledge the generous support of An Foras Feasa, who funded this research.
References


