Introducing *Parselmouth*: A Python Interface to *Praat*

Yannick Jadoul[a,b,*], Bill Thompson[c,a], Bart de Boer[a,c]

[a] *Artificial Intelligence Lab Brussels, Vrije Universiteit Brussel*
Pleinlaan 2, 1050 Elsene, Belgium
[b] *ADReM Research Group, University of Antwerp*
Middelheimlaan 1, 2020 Antwerpen, Belgium
[c] *Language and Cognition Department, Max Planck Institute for Psycholinguistics, Wundtlaan 1, Nijmegen 6525 XD, The Netherlands*

Abstract

This paper introduces *Parselmouth*, an open-source Python library that facilitates access to core functionality of Praat in Python, in an efficient and programmer-friendly way. We introduce and motivate the package, and present simple usage examples. Specifically, we focus on applications in data visualisation, file manipulation, audio manipulation, statistical analysis, and integration of Parselmouth into a Python-based experimental design for automated, in-the-loop manipulation of acoustic data. Parselmouth is available at `https://github.com/YannickJadoul/Parselmouth`.

Keywords: Praat, Python, Data Analysis, Acoustics, Phonetics, Software

1. Introduction

Data analysis in the phonetic sciences routinely relies upon the functionality of *Praat* (Boersma, 2001; Boersma & Weenink, 2018), an extensive software package which has subserved the day-to-day activities of phoneticians for more than two decades. This paper introduces *Parselmouth*, an open-source Python library that exposes major functionality of Praat into Python. Two principal advantages result from this integration: 1) users of Praat may now benefit from the expressive power of a large-scale language like Python, and its expansive ecosystem of scientific and computational libraries; and 2) users of Python may access the many tools and utilities for sophisticated acoustic analysis that Praat provides. Parselmouth is currently available as version 0.3.0, for use with Windows, macOS, and Linux-based operating systems, for Python versions 2 and

*Corresponding author
Email address: `Yannick.Jadoul@ai.vub.ac.be` (Yannick Jadoul)

3. The package is under active development by the first author of this article, and can be downloaded from `https://github.com/YannickJadoul/Parselmouth`. Basic speech analysis methods from Praat are already available, while all other algorithmic functionality of Praat can be called indirectly. We are hopeful that others in the community of speech scientists and engineers will wish to contribute to the development.

The remainder of this paper is organised as follows: Sections 1.1, 1.2, and 1.3 respectively give detailed background and motivation, compare Parselmouth to other software packages, and provide technical information about Parselmouth. Sections 1.4, 1.5, and 1.6 then present more practical information on the functionality of Parselmouth, how to install the Python library, and where to find its online documentation and further resources. Section 2 presents five usage examples, focusing on what we imagine to be some of the most recurrent technical challenges speech scientists are likely to face: idiosyncratic visualisation of acoustic data (Section 2.1); reading, writing, and manipulating batches of acoustic files and data frames (Section 2.2); manipulation of audio files along complex acoustic dimensions (Section 2.3); statistical analysis of the output of acoustic analyses (Section 2.4); and integration of automated acoustic analysis into experimental design (Section 2.5). These examples are intended to be illustrative of the principles behind the package, rather than exhaustive demonstrations of Parselmouth's potential use cases, which we expect to grow indefinitely with the advance of Python and the scientific creativity it facilitates. We summarise the motivation and examples and present concluding remarks in Section 3, after which Section 3.1 closes the paper with a brief discussion of the future of Parselmouth.

At this point we wish to stress that Parselmouth is built on the vast Praat collection of source code: as such, we encourage twin citation of both Praat and Parselmouth whenever Parselmouth is used for scientific research.

1.1. Motivation

The Python programming language is rapidly becoming the *lingua franca* of scientific computing. Python is used and supported by an enormous community of scientists, researchers, and engineers whose workflows are continuously improving thanks to integration of diverse computational utilities in a single programming language. For many, including us, Python is the go-to toolbox for data manipulation and analysis. However, for contemporary *speech* scientists, researchers, and engineers, major portions of our day-to-day activities – specifically, analysis of acoustic data using Praat functionality – remain difficult or time-consuming in Python; the necessary functionality is often unavailable or dispersed over multiple unrelated and sometimes incompatible libraries. We began developing Parselmouth as a solution to this problem. Parselmouth is not a replacement for Praat: it is an additional interface to Praat, making Praat's functionality available in Python. We have three principal goals in mind: to allow experienced users of Praat to more efficiently integrate acoustic analysis with scientific tools available in Python but not in Praat; to provide access to Praat's functionality for users who are comfortable with Python but unfamiliar with Praat; and to simplify or optimise the workflow of any users who would simply rather work in a single language.

Python is often used as *glue language* for scientific workflows, drawing together the "scientific stack" in a collection of widely used, robust scientific libraries (e.g., *NumPy*, *SciPy*, *pandas*, *scikit-learn*, *matplotlib*, etc.; see `https://scipy.org/about.html`). As

Python is designed as an extensible programming language and framework, its use extends across many domains, and even across other programming languages. Scientists using Python have access to, for example: advanced statistical modeling libraries and probabilistic programing frameworks such as *Statsmodels*[1], *PyMC3*[2], *Pyro*[3], and *Edward*[4]; deep learning libraries like *TensorFlow*[5] or *PyTorch*[6]; *Jupyter* notebooks[7] (formerly IPython); experimentation packages such as *PsychoPy*[8] or *Dallinger*[9]; the *rpy2*[10] module that provides easy access to R functionality; and the official *'MATLAB Engine API for Python'*[11], which integrates MATLAB into Python programs. More generally, just like Praat, Python has functionality for writing universal data exchange formats – built-in, such as comma-separated values (csv) or JavaScript Object Notation (JSON), or through external libraries, such as HDF5[12] or SQL databases[13,14] – which makes it possible and convenient to use Parselmouth to combine the functionality of Praat and these Python libraries with almost any other computational framework.

While choosing any particular language is to some extent an arbitrary choice, Python is a popular and high-level, yet fully-fledged programming language. Python not only accommodates quick scripting but also provides support for more complex programming paradigms and performant implementations of algorithms. While the Praat scripting language is suitable for automating repeated workflows and calculations within the context of Praat, we believe the use of Python and Parselmouth can be advantageous in a broader range of applications. Python implements general programming principles, including a full and generic type system with built-in types (i.e., lists, tuples, sets, dictionaries, ...) and custom classes. As a result, Python is well-suited to be used in a more programming-intensive context. In these cases, *integrated development environments* with e.g. syntax highlighting, autocompletion functionality, and debugging tools, can assist in the development process.

Python is also an accessible language, useful for writing simple scripts. Python is often taught to students at their first encounter with programming, sometimes even before a specialisation in phonetics brings them into contact with Praat. We believe that Parselmouth can be attractive to this group of users that are already familiar with programming or Python, but not with the Praat scripting language. Python is supported by a large community of users who have written up many solutions to specific programming problems and frequent Python errors – see, for example, StackOverflow[15]. Fewer people have the necessary experience with Praat to answer questions and solve problems con-

[1] `http://www.statsmodels.org/`
[2] `http://docs.pymc.io/`
[3] `http://pyro.ai/`
[4] `http://edwardlib.org/`
[5] `https://www.tensorflow.org/`
[6] `http://pytorch.org/`
[7] `http://jupyter.org/`
[8] `http://www.psychopy.org/`
[9] `http://docs.dallinger.io/en/v3.4.1/`
[10] `https://rpy2.readthedocs.io/`
[11] `https://mathworks.com/help/matlab/matlab-engine-for-python.html`
[12] `http://docs.h5py.org/`
[13] `http://www.sqlalchemy.org/`
[14] `https://pandas.pydata.org/pandas-docs/stable/io.html#io-sql`
[15] `https://stackoverflow.com/questions/tagged/python`

cerning Praat scripts, and fewer resources and tutorials exist to learn the Praat scripting language than to learn Python.

Finally, the Python project and the available libraries are *modular*. They are specialised in one area of functionality (i.e., being a programming language, plotting graphs, handling data tables, performing statistical analyses, etc.), yet are designed to be used and combined in larger and more complex projects. With Parselmouth, we aim to add the option of using the highly advanced, specialised functionality from Praat in combination with the already existing libraries in Python in this same manner.

1.2. Relation to previous software

Parselmouth is not the first attempt to port Praat functionality into Python. Other packages exist, together offering a range of Praat functionality. However, the previous projects we are aware of are generally restricted in important ways that Parselmouth is not, technically speaking. We see the diversity of preceding projects as testament to a clear but unfulfilled demand for sophisticated acoustic data analysis tools in Python. We are aware of *praat-py*[16], *praat-python-scripts*[17], *praatIO*[18], and *textgrid*[19].

Generally speaking, we found two approaches in these projects. On the one hand, some projects *reimplemented* a selection of Praat's functionality in Python code. A significant drawback of this approach is that it does not guarantee the same results as Praat, due to the potential for subtly distinct implementations, and the possibility of introducing errors in newly-written code. This approach requires a great deal of effort for every extra bit functionality, and requires familiarity with and insight into the phonetic algorithms that Praat has already implemented, tested, and refined. Other existing projects instead provide a Python interface to Praat scripts and its scripting language. This often has the disadvantage of compromised performance because of increased communication between Python and a separate Praat program (cfr. Section 1.3). Moreover, a Python user still needs to learn the Praat scripting language, rather than being able to use a *"pythonic"*[20] Python library.

While all these projects fulfill the needs of the contexts in which they were created, and might certainly be used and combined by other users, none of these satisfactorily provides efficient access to a broad range of Praat functionality in a pure Python environment. Parselmouth combines the strengths of these approaches to offer a fully pythonic Python library – i.e. classes, functions, operators, etc. that look and function just like other familiar libraries in Python. This is made possible by a more comprehensive technical solution to the challenge of linking Python and Praat's code. Rather than re-implementing the complex algorithms underpinning Praat, Parselmouth utilises Praat's own official C/C++ (open) source code behind the scenes. This ensures that any analyses conducted using Parselmouth are completely consistent with Praat, without the

[16] `https://github.com/JoshData/praat-py`

[17] `https://github.com/mmcauliffe/python-praat-scripts`

[18] `https://github.com/timmahrt/praatIO`

[19] `https://github.com/kylebgorman/textgrid`

[20] *"To say that code is pythonic is to say that it uses Python idioms well, that it is natural or shows fluency in the language, that it conforms with Python's minimalist philosophy and emphasis on readability."* – `https://en.wikipedia.org/wiki/Python_(programming_language)` (accessed 14th of September, 2017)

user needing to know Praat's scripting language. Moreover, by reaching directly into the source code of Praat, Parselmouth's access to Praat's data structures and routines is fast and efficient. Section 1.3 gives a very brief description of how this works and why we believe our solution achieves a desirable trade-off between the available approaches.

In addition, numerous scientific audio tools and libraries are available to users of Python (see, e.g., `https://github.com/faroit/awesome-python-scientific-audio`): we hope that Parselmouth can complement these tools, whilst also providing a unified suite of tried-and-tested routines for analysis of speech data, specifically in Praat.

We are also aware of libraries that allow access to Praat functionality for the R language and MATLAB environment: *PraatR* (Albin, 2014)[21], and *rPraat* and *mPraat* (Bořil & Skarnitzl, 2016)[22]. We see these packages as complementary to Parselmouth: their availability caters to the needs of users of these other languages, but does not provide a convenient solution for the many Python users in the scientific community. Our impression is that these packages are generally subject to the same restrictions as the Python packages we reviewed above: either they provide access to Praat functionality, but only by calling Praat commands externally (*PraatR*), or they *re-implement* a subset of Praat (*mPraat* & *rPraat*).

More generally, we are currently witnessing an exciting expansion of digital tools for open and collaborative manipulation, management, and analysis of speech data. One project in this vein is the EMU Speech Database Management System[23], which aims to harness digital tools to expand the range of speech-data annotation and indexing capabilities currently available to speech scientists. We hope that Parselmouth can be understood as a small contribution to this general movement to increase the accessibility of speech-analysis methods.

1.3. Technical details

The official Python C API[24] makes it possible to use the compiled C/C++ routines from Praat directly in Python. In particular, Parselmouth relies on the pybind11 library (Jakob et al., 2017) for low level, efficient communication with and access to Praat's internal objects, memory, and code. This makes Parselmouth fast and efficient by removing the need to send large lists and grids as strings of numbers (which would first have to be serialized, then parsed, etc.) between programs. Because Praat is part of the Parselmouth library instead of using an external version of the Praat program, we can provide immediate access to the raw data calculated by Praat. Moreover, *NumPy* (Walt et al., 2011) allows us to directly use data rather than making an entire copy. Consider calculating a spectrogram for one second of audio, for instance, using Praat's default time step of 0.002 s and maximum frequency of 5000 Hz. This would result in roughly 500 time slices that all consist of 160 frequency bins, or about 80000 floating point values in total. When Praat calculates these values, they already exist in a 2D array, stored in memory. Parselmouth together with *NumPy* lets you use the existing values without copying, rather than copying all of them into Python lists or making 80000 calls into

[21] `http://www.aaronalbin.com/praatr/`
[22] `http://fu.ff.cuni.cz/praat/`
[23] `http://ips-lmu.github.io/EMU.html`
[24] `https://docs.python.org/3/c-api/index.html`

a running Praat instance to request the values. And in the case that all of these numeric values would have to be converted back and forth to string representations during inter-process communication with the `sendpraat` tool or subprocess calls to the Praat executable, the efficiency would even decrease more dramatically.

As such, Parselmouth's performance is notably fast. On the one hand, when it comes to the execution of Praat's functionality, we are using the exact same code, and Python scripts that access computationally expensive Praat algorithms are expected to take the same amount of time. On the other hand, when it comes to the comparsion of Praat and Parselmouth scripts that have a high rate of interaction between the Python code and the Praat functionality, our tests and benchmarks seem to indicate that the combination of Python and Parselmouth runs as fast or even faster than the equivalent script runs in the Praat interpreter. Consequently, the necessary conversions and communication between Python and Praat do not seem to make Parselmouth less efficient than using Praat scripts. A few pairs of equivalent Python and Praat scripts can be found in the documentation and supplementary material for a comparison in performance.

As is often the case in software development, the technical solution is a trade-off between multiple, often conflicting goals and considerations. One of the disadvantages of Parselmouth's solution is the fact that the C/C++ code results in platform-dependent versions of the library (unlike a pure Python library). However, given the existence of Python standards related to the compatibility of binary libraries[25,26,27], and the important advantage of reusing Praat's existing code, we consider this a suitable compromise. Moreover, many other scientific Python libraries – such as *Numpy*, *SciPy*, and *pandas* – have made a similar decision because of code reuse and performance reasons.

Another trade-off in Parselmouth is the manual creation of a completely new "pythonic" interface, rather than automatically converting the Praat commands. While this means that the actual development and maintenance of the project is more labour-intensive, this allows us to create a Python-intuitive library that integrates well with common Python idioms and other libraries such as *NumPy*. Moverover, the complementary `praat.call` and `praat.run` functions (cfr. Sections 2.3 and 2.4) do provide immediate access to the full scope of Praat functionality, independently from the pythonic Parselmouth interface. This choice is related to our decision of including a specific version of Praat as part of the Parselmouth Python package. This sacrifices the flexibility of picking a specific version of Praat in favour of the possibility to access Praat's internal values and structures. The latter makes Parselmouth more efficient in use, easier to install, and allows us to fine-tune the Python interface. Moreover, since multiple versions of Praat are designed to be compatible, we consider this to be the desired trade-off for Parselmouth.

1.4. What Praat functionality is already ported in Parselmouth?

Parselmouth currently supports 8 classes: `Sound`, `Spectrum`, `Spectrogram`, `Intensity`, `Pitch`, `Formants`, `Harmonicity`, and `MFCC` (and conversion between these objects). We are in the process of porting the `TextGrid` functionality. In addition to this

[25] `https://www.python.org/dev/peps/pep-0491/`
[26] `https://www.python.org/dev/peps/pep-0513/`
[27] `https://www.python.org/dev/peps/pep-0571/`

primary functionality, which has been designed to look and feel like a native and efficient Python library, Parselmouth also implements access to the Praat commands and scripts. This way, a user can access familiar Praat functionality that has not (yet) been explicitly added to Parselmouth, as we demonstrate in Sections 2.3 and 2.4. We are also keen to learn from the research community the aspects of Praat that would be in high demand in a further development of Parselmouth. We discuss future functionality in Section 3.1.

1.5. Installation

Parselmouth is available in the Python Package Index (PyPI)[28] under the name `praat-parselmouth`[29] and can be installed via the default package manager `pip` using the following command:

```
pip install praat-parselmouth
```

Parselmouth's source is hosted on GitHub at `https://github.com/YannickJadoul/Parselmouth`. Updates, examples, and troubleshooting advice can also be found in this repository. The accompanying documentation (cfr. Section 1.6) provides up-to-date details on the installation and includes instructions on how to install Parselmouth in the PsychoPy Builder interface.

We also intend to keep Parselmouth up to date with the newest Praat updates: at the time of writing the current version, 0.3.0, is based on Praat 6.0.37 (the version of Praat released on the 3rd of February, 2018).

1.6. Documentation

Automatically generated documentation, advanced installation instructions, and more usage examples are available at `https://parselmouth.readthedocs.io`. We are constantly improving the documentation and usage examples, and we are hopeful that others in the community will wish to contribute to this effort. To help out fellow scientists and Parselmouth users, we highly appreciate feedback on the current state of Parselmouth and the documentation, the identification of potential problems, the completion of an example, or the addition of a usage tutorial.

2. Usage Examples

We provide five simple usage examples that focus on integration of Parselmouth into common Python-based workflows. This focus reflects our assumption that the used Praat functionality, which Parselmouth simply uses directly, is already familiar to the user. Users who are not familiar with Praat's functionality can find excellent tutorial examples in Praat's documentation. The usage examples we provide are intended to demonstrate the basic principles and efficiencies of using Praat functionality in a Python workflow, rather than to be examples of tasks that Praat could not accomplish per se. We wish to explicitly acknowledge the flexibility of Praat in this respect here. Parselmouth simply

[28] `https://pypi.python.org/pypi`

[29] Do please note that while the Python module itself is called `parselmouth`, the PyPI package to install with `pip` is `praat-parselmouth`, and *not* the unrelated `parselmouth` package.

provides an alternative means of interaction with Praat's algorithms, which we hope can be beneficial to some users. The code in these examples is made available as part of the Parselmouth repository at `https://github.com/YannickJadoul/Parselmouth` and the documentation at `https://parselmouth.readthedocs.io`, and in the supplementary materials for this article. The supplementary material also contains an annotated version of all examples that describes in detail what the Python code is doing.

Rather than presenting a single case study in which we demonstrate the use of Parselmouth, we choose to present short and independent examples – with a limited amount of code and minimal complexity – to demonstrate the variety of applications, practically motivate the project, and hopefully inspire the use of Parselmouth in concrete research. After all, Praat itself is used to study a wide range of research questions and can be used as a tool in different phases of that research, and it seems inopportune to narrow down the use of Praat and Parselmouth to a single example. Moreover, Python is a programming language, a tool to write scripts and programs that create new and modify custom-tailored workflows. Abstract and short examples can give an idea of the range of situations in which Parselmouth could be useful, leaving the broader context up to the user to define. We are however hopeful that over time, more complex and more concrete examples can be added to the documentation, and that new research can demonstrate the applicability of Parselmouth (e.g., Ravignani, 2018).

These examples are chosen to represent different parts of a hypothetical phonetic experiment. First, acoustic stimuli need to be created and played to participants (Sections 2.3 and 2.5). Afterwards, some post-processing of the collected data is required (Section 2.2), and finally the results can be plotted and subjected to statistical analyses (Sections 2.1 and 2.4, respectively). While these different abstract examples need to be seen in the context of a larger scientific workflow, to be adapted to match one's specific needs in a concrete project, we present these examples out of order for educational purposes: each of the examples is practically independent of the previous ones but will build further upon the concepts introduced before.

2.1. Data Visualisation

Effective visualisation of acoustic data is an art form. Seamless generation of professional-looking and highly accurate spectrograms has always been one of Praat's major attractions. Parselmouth is not intending to replace or supersede Praat's visualisation routines, which are finely tailored for human speech data: for quick and easy spectrograms for instance, Praat remains the better option (in our view). However, Parselmouth allows the computation of a phonetic analysis to be more easily separated from the choice of a framework for visualisation and presentation. While most, if not all, common visualisation needs can be fulfilled with Praat's Picture window[30], Parselmouth's modularity allows a user to access more exotic plot types and features of different Python graphing packages, to combine the plots with custom statistical insights and plots that might not be available in Praat, to have the plots shown in a Jupyter notebook, or maybe just to use existing experience in Python visualisation that the user does not possess in Praat.

[30] `http://www.fon.hum.uva.nl/praat/manual/Picture_window.html`

The example in Listing 1 shows two simple Python functions that integrate Parselmouth and the Python visualisation libraries *matplotlib*[31] (Hunter, 2007) and *seaborn*[32] in order to plot a colourful spectrogram and an overlaid pitch contour. Notice how the calls to Praat functionality and information through Parselmouth (e.g. `sound.to_spectrogram()`, `spectrogram.values`, or `pitch.ceiling`) are integrated within the Python logic, rather than isolated calls into Praat. Though obtaining and plotting the values of a spectrogram in Python is in itself not a difficult challenge with the help of existing libraries (*matplotlib*, for example, includes a function `specgram`, and *SciPy*'s `signal` module contains `spectrogram`), we are using Praat's tried and tested algorithm to calculate the spectrogram's values. This means that the same set of familiar parameters from Praat can be used, and more importantly that this will result in the exact same analysis you would get in Praat.[33]. Figure 1 shows the resulting plot.

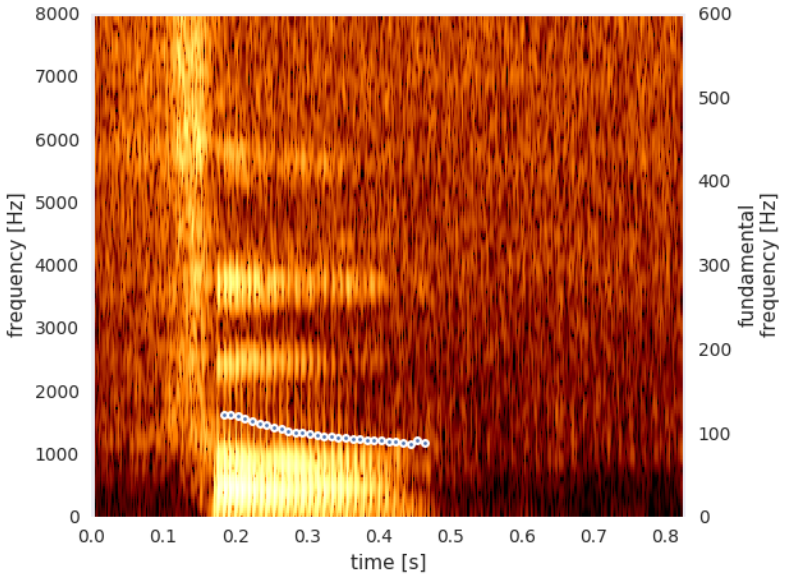

Figure 1: Custom spectrogram and pitch contour resulting from the Python code in Listing 1.

Apart from the actual visualisation example and the combination of Praat functionality with the *matplotlib* library, the main thing to take away from Listing 1 is the mapping between Praat and Parselmouth objects and functionality. After defining the two auxiliary functions `draw_spectrogram` and `draw_pitch`, we open an audio file as `Sound` object, just as one would do in Praat. The main difference is that we store the

[31] `https://matplotlib.org/`

[32] `http://seaborn.pydata.org/`

[33] Beware however that assuming manual control over the plotting does mean that you need to watch out to not make mistakes that Praat's standard plotting algorithms avoid and abstract away from the user. For example, the exact timing of the spectrogram samples with respect to the time range of the sound signal are automatically handled by Praat but require in our example to respectively use `xs()` (or `x1`, `nx`, and `dx`) vs. `xmin` and `xmax`.

```python
import parselmouth

import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set() # Use seaborn's default style to make attractive graphs

def draw_spectrogram(spect, dynamic_range=70):
    X, Y = spect.x_grid(), spect.y_grid()
    sg_db = 10 * np.log10(spect.values)
    min_db = sg_db.max() - dynamic_range
    plt.pcolormesh(X, Y, sg_db, vmin=min_db, cmap='afmhot')
    plt.ylim([spect.ymin, spect.ymax])
    plt.xlabel("time [s]")
    plt.ylabel("frequency [Hz]")

def draw_pitch(pitch):
    # Extract selected pitch contour, and
    # replace unvoiced samples by NaN to not plot
    pitch_values = pitch.selected_array['frequency']
    pitch_values[pitch_values==0] = np.nan
    plt.plot(pitch.xs(), pitch_values, 'o', markersize=5, color='w')
    plt.plot(pitch.xs(), pitch_values, 'o', markersize=2)
    plt.grid(False)
    plt.ylim(0, pitch.ceiling)
    plt.ylabel("fundamental frequency [Hz]")

snd = parselmouth.Sound("audio/4_b.wav")
pitch = snd.to_pitch()
# Optionally pre-emphasize the sound before calculating the spectrogram
snd.pre_emphasize()
spectrogram = snd.to_spectrogram(maximum_frequency=8000.0)

plt.figure()
draw_spectrogram(spectrogram)
plt.twinx()
draw_pitch(pitch)
plt.xlim([snd.xmin, snd.xmax])
plt.show()
```

Listing 1: Using Parselmouth to plot the custom spectrogram visalisation in Figure 1. Usage of Parselmouth functionality is highlighted in red; a version with detailed comments can be found in the supplementary material.

object in a variable `snd` rather than adding it to the global list of objects in Praat. Afterwards, our invocation of `snd.to_pitch()` corresponds to first selecting the `Sound` in Praat's objects list and then clicking `To Pitch` in the Praat user interface (or writing the equivalent script). As we have designed Parselmouth with this mapping in mind, this is also how other objects and Praat commands are accessible through Praat: Praat objects become standard Python objects in variables, and Praat commands become methods of these objects. Notice this principle being applied, for example, in `spectrogram = snd.to_spectrogram(maximum_frequency=8000.0)`: `snd` is a Praat/Parselmouth `Sound` object, `To Spectrogram` is called for this selected object, one parameter of this call is changed to a non-standard value (`maximum_frequency=8000.0`), and the `Spectrogram`

object Praat creates and would add to the global list of objects is returned and stored in the variable `spectrogram`. A user interested in e.g. the harmonics-to-noise ratio rather than the fundamental frequency can now correctly infer that the Parselmouth equivalent of `To Harmonicity` is the `Sound.to_harmonicity` method[34].

Listing 2 shows how this kind of plotting function can be combined with the Python data manipulation library *pandas* and the `FacetGrid` functionality of *seaborn* to compose a structured array of spectrograms with overlaid pitch contours. This example visualises a small dataset consisting of the numbers 1 to 5 being spoken in English by the first two authors. The example assumes we have stored these audio files in a directory named `audio`, and that each audio file has been named in accordance with the convention `{digit}_{speaker-id}.wav`. It also assumes a csv data frame whose rows contain variables that uniquely identify a speaker-id / digit combination that we wish to plot in the grid. The next example (see Section 2.2) goes into more detail on file system integration for structured data frames. The resulting array of spectrograms, with the number being spoken along the columns, and a row for each speaker, is shown in Figure 2.

```
import pandas as pd

def facet_util(data, **kwargs):
    digit, speaker_id = data[['digit', 'speaker_id']].iloc[0]
    sound = parselmouth.Sound("audio/{0}_{1}.wav".format(digit, speaker_id))
    pitch = sound.to_pitch()
    sound.pre_emphasize()
    draw_spectrogram(sound.to_spectrogram())
    plt.twinx()
    draw_pitch(pitch)
    # If not the rightmost column, then clear the right side axis
    if speaker_id != 'y':
        plt.ylabel("")
        plt.yticks([])

results = pd.read_csv("audio/digit_list.csv")

grid = sns.FacetGrid(results, row='digit', col='speaker_id')
grid.map_dataframe(facet_util)
grid.set_titles(col_template="{col_name}", row_template="{row_name}")
grid.set_axis_labels("time [s]", "frequency [Hz]")
grid.set(xlim=(0, None))
# Optionally: grid.set(facecolor='white')
plt.show()
```

Listing 2: Plotting a data set as custom spectrograms (see Figure 2). Usage of Parselmouth functionality is highlighted in red; a version with detailed comments can be found in the supplementary material.

In the example code in Listing 2, there is one more aspect related to the usage of Parselmouth that deserves focus. The function `facet_util` is repeatedly called by *seaborn*'s `FacetGrid` visualisation of the audio files in a grid layout, but in the implementation of `facet_util` this does not matter: Parselmouth is used to access specific

[34]For an overview of the available objects and methods, consult Python's built-in `help` function (e.g., `help(parselmouth.Sound)`) or the API reference section of the documentation.

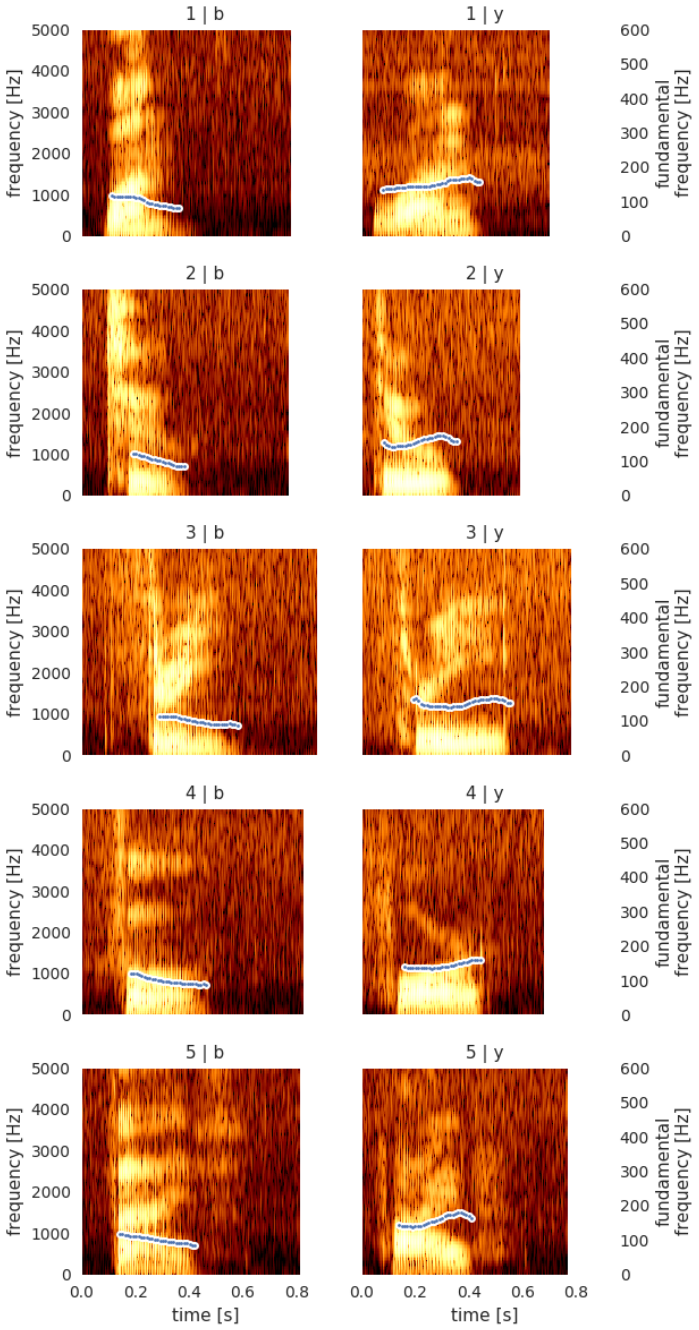

Figure 2: Structured arrangement of custom spectrograms (cfr. Listing 2).

condition	···	**pp_id**
0	···	1877
1	···	801
1	···	2456
0	···	3126

Table 1: Example structure of a csv file with results to be analysed by the code in Listing 3.

Praat functionality *when logically needed* by the program. This means that there is no need to do the analysis before the plotting, for example, as one might do when using Praat and a Python script separately. Again, this programming pattern can be applied in different contexts from our simple example; the actual Praat functionality accessed might be different from the pitch estimation and spectrogram calculation done here. Next to the demonstration of *matplotlib* or *seaborn* plotting, the idea to take away from this example is that Parselmouth enables a user to use Praat functionality at the location it is logically needed within a Python script or program. Moreover, notice how, just like in Praat, we can reuse existing code by calling previously defined function (here, `draw_spectrogram` and `draw_pitch` from Listing 1). As a general principle, the Parselmouth objects (here, `sound`, `pitch`, and the return value of `sound.to_spectrogram()`) are ordinary Python objects and can be stored in variables and passed to existing functions.

2.2. File Manipulation

Python and its libraries provide rich structures and utilities for storage and manipulation of structured datasets. When combined with Parselmouth, these tools can facilitate painless acoustic analysis of arbitrarily complex datasets. Consider the following workflow: the phonetician performs an experiment which results in multiple participants in multiple experimental conditions, producing audio files from which the mean harmonics-to-noise ratio needs to be extracted. A typical scenario might result in audio files being stored on the phonetician's computer in subdirectories of a directory named `results`. These subdirectories reflect the different experimental manipulations, and each audio file is named with a unique participant identifier number. The experiment also resulted in a large csv file whose rows provide information on each participant, including variables that indicate the experimental manipulation (*condition*) and the unique participant identifier (*pp_id*). Table 1 shows a simplified example of this structure.

In this example, we show how, in tandem with the data manipulation library *pandas*[35] (McKinney, 2010), Parselmouth makes light work of the data analysis task: looping over a large dataset, identifying an appropriate audio file in the user's file system, extracting the harmonics-to-noise-ratio of the audio at a certain time as a single Python decimal number, and writing this value back into the appropriate row of the results data frame. The extract below in Listing 3 shows how this operation can achieved in just a few lines of Python code using Parselmouth. Once again, this example serves as a general illustration of the kind of batch processing one could do on a set of audio files in order to, for example, extract a certain set of acoustic features (e.g., Ravignani, 2018).

[35] `https://pandas.pydata.org/`

```python
import parselmouth
import pandas as pd

def analyse_sound(row):
    condition, pp_id = row['condition'], row['pp_id']
    filepath = 'audio/{}_{}.wav'.format(condition, pp_id)
    harmonicity = parselmouth.Sound(filepath).to_harmonicity()
    return harmonicity.get_value(row['time'])

# Read in the experimental results file
dataframe = pd.read_csv('results.csv')

# Apply parselmouth wrapper function row-wise
dataframe['harmonics_to_noise'] = dataframe.apply(analyse_sound, axis='columns')

# Write out the updated dataframe
dataframe.to_csv('processed_results.csv', index=False)
```

Listing 3: Example analysis on structured data (such as Table 1), using the combination of *pandas* and Parselmouth to manipulate the csv data frame and perform a custom acoustic analysis. Usage of Parselmouth functionality is highlighted in red; a version with detailed comments can be found in the supplementary material.

The way Parselmouth is used here is very similar to its usage in Listings 1 and 2 in Section 2.1: an audio file gets loaded as `Sound` object and subjected to an acoustic analysis. However, rather than using all values calculated by Praat (to visualise them), we here have Praat interpolate the value at specific points in time. In this example, we chose to include the times' offsets at which the harmonics-to-noise ratio is extracted in the comma-separated values file. Alternatively, a common scenario would be where each audio file is accompanied by a Praat `TextGrid` with annotations. Querying a `TextGrid` file or object through Parselmouth is illustrated and explained in the example in Section 2.4.

A related and common use case is manipulating or analysing all files in a certain directory or whose name matches a certain pattern. Analogous to Praat's `Create Strings as file list...`[36], Python's built-in `glob.glob` function[37] allows one to find and loop over these files in a single line (i.e., `for file_name in glob.glob("subdir/*.wav"):`). While this is possible to do in Praat, we imagine the ease of combining this construct with other examples and workflows to be attractive and useful to Python and Parselmouth users.

2.3. Audio Manipulation

In addition to the pythonic interface at the core of this project, Parselmouth also provides access to Praat's functionality by means of calling the commands visible in the user interface, as one would do in a Praat script. This offers two advantages: 1) functionality not yet ported to the core interface can still be accessed in this way; and 2) users who already have a strong working knowledge of Praat menu commands and buttons

[36] `http://www.fon.hum.uva.nl/praat/manual/Create_Strings_as_file_list___.html`
[37] `https://docs.python.org/3/library/glob.html`

can call those methods by name if they choose. Listing 4 lays out an example relating to the manipulation of the pitch track of an existing audio recording. Manipulating the fundamental frequency of an audio file is a complicated procedure, and unlikely to be readily available in Python-based tools, yet the requirement for close control over the pitch of acoustic experimental stimuli can be used for testing questions related to the effect of sound manipulation on language acquisition and perception (see e.g., Filippi et al., 2014). In the example set out in Listing 4, we access functionality related to Praat's `Manipulation` class and call a number of Praat actions to increase the fundamental frequency of an audio sample by one octave.

```
import parselmouth
from parselmouth.praat import call

sound = parselmouth.Sound("audio/4_b.wav")

manipulation = call(sound, "To Manipulation", 0.001, 75, 600)
pitch_tier = call(manipulation, "Extract pitch tier")

call(pitch_tier, "Multiply frequencies", sound.xmin, sound.xmax, 2)

call([pitch_tier, manipulation], "Replace pitch tier")
sound_octave_up = call(manipulation, "Get resynthesis (overlap-add)")

sound_octave_up.save("4_b_octave_up.wav", "WAV")
```

Listing 4: Code extract reading in an audio file and directly using Praat commands to increase the fundamental frequency of the audio fragment by one octave. Note how the code uses the `parselmouth.praat.call` function, since the `Manipulation` and `PitchTier` classes are currently not yet available as ordinary Python objects in Parselmouth. Usage of Parselmouth functionality is highlighted in red; a version with detailed comments can be found in the supplementary material.

The example in Listing 4 demonstrates two things: functionally, it shows how to access the `Manipulation` functionality in Praat, but more importantly, it demonstrates how to use Parselmouth's `call` function. As argued above, changing the pitch of an audio fragment is a non-trivial task that is easily achievable in Praat, and integrating this into a larger Python context might be a reason to use Parselmouth. However, with this concrete example, we also want to demonstrate how to use `praat.call` by showing the one-to-one mapping to the Praat user interface and scripting language. The first (optional) argument is a Parselmouth object or a list of objects; these are the objects that would be selected in Praat when executing the command. Next, the name of the Praat button or action is passed as an argument to `call`, and after that the arguments for the action are listed (i.e., the values one would type in the Praat form or write in a Praat script). Parselmouth takes care of converting the arguments to Praat types and returning the result of the Praat action as a Python type or Parselmouth object.

Also note how writing `intensity = call(sound, "To Intensity", 100.0, 0.0, False)` and `intensity = sound.to_intensity(subtract_mean=False)` are equivalent. The former passes through the Praat command interface, while the latter is uses the pythonic Python interface Parselmouth provides. The advantage of the latter approach is that it only requires arguments that are different from Praat's default values to be

specified, fits better with standard Python coding styles and convention, and is internally slightly more efficient. However, both approaches access the same underlying Praat code, and as such `praat.call` can be used when preferred or when the main Python interface is not present in the current version of Parselmouth.

Lastly, while the example in Listing 4 is kept simple and abstract and is thus probably not directly applicable in a specific use case, we want to suggest combining it with the principles outlined in the other examples. For instance, the combination with the file manipulation example from Section 2.2 would allow one to automate the pitch manipulation for multiple files. Another possibility is integrating this code into the interactive experiment, as we will demonstrate in Section 2.5.

2.4. Integration with Statistical Libraries & Existing Praat Scripts

Another advantage of the Parselmouth workflow is integration of acoustic and statistical analysis in one language and software environment. Although statistical analysis is possible in Praat too, the range of statistical analysis libraries available to users of Python is vast, robust, and adapts reliably to incorporate new methods, because of the modular structure of the Python environment: i.e., a user can install new libraries for the necessary statistical analyses. This example demonstrates a simple workflow integrating these two forms of analysis. We also use this example to illustrate Parselmouth's ability to execute Praat functionality by calling Praat scripts. While the primary purpose of Parselmouth is to provide a pythonic Python API for Praat, we recognize that access to Praat functionality through this interface is currently limited to a subset of all available Praat functionality. Parselmouth includes the ability to invoke Praat functionality by calling Praat commands directly (akin to the way previous libraries handled integration) as a solution for this limitation, while the Parselmouth codebase grows (cfr. Section 2.3). This flexibility also caters for another potentially common use case: execution of already existing, *legacy* Praat scripts.

The usefulness of running Praat scripts from Python becomes clear in a scenario where one wants to reuse previously written Praat scripts that perform some sophisticated acoustic analysis, to for example apply such a script to a new dataset. When the user would also wish to run some heavy-duty or uncommon statistics on the results using a specialized statistical library, and has learned how to do so in Python, Parselmouth can be used to integrate the entire process into one workflow. Rather than tediously re-writing the existing Praat script in Python, Parselmouth allows the user to run the Praat script from Python and to interface the input and output of the script with the rest of the Python code. We envisage this kind of scenario to be relatively common as a substantial amount of research has already been done using Praat and Praat scripts.

In this example, presented in Listings 5 and 6, we show how Parselmouth can be used to execute a Praat script by De Jong & Wempe (2009) for automatic extraction of syllable centers, and to subsequently perform statistical analysis. In Listing 5, we apply this existing Praat script to a corpus of audio recordings of Aesop's fable *The North Wind and the Sun* in different English dialects or accents, openly available from the LibriVox project[38]. The example then shows, in Listing 6, how to feed the results of the acoustic analysis directly into a statistical analysis of the syllable centre time series data. In this

[38] `https://librivox.org/celebration-of-dialects-and-accents-vol-1/`

case, we use a mixed-effects linear model from two different Python statistics libraries, StatsModels[39] and BAMBI[40] (BAyesian Model-Building Interface, based on PyMC3[41]).

In this short example, we test the null hypothesis that readers with a native accent read aloud the story equally fast as the non-native readers. While the corpus used in the example is arguably rather small, and the mixed-effects linear model being fitted might not be the optimal statistical method, we merely use this statistical question and approach as a simple demonstration of how one would combine a Praat analysis and Python statistical analysis in a single workflow. Our motivation to present such an example stems from a past project where we have examined a similar corpus of recordings of *The North Wind and the Sun* (International Phonetic Association, 1999) with regard to syllable timing predictability (Jadoul et al., 2016), using an *autoregressive integrated moving average* (or ARIMA) model for time series analysis.

In Listing 5 the first part of this process is demonstrated. The `extract_syllable_intervals` function calls the existing Praat script through `praat.run_file`. Just like the `call` function, `run` and `run_file` take an optional Praat object or list of objects as first argument to be selected when at the start of the script's run; in this case, we omit the argument since it is not necessary for our example. After the file name of the script, we pass the further arguments to the script; again, this is analogous to running the script in Praat, where one would get a window with the parameters to the script declared in a `form`-construct in the script[42]. Parselmouth takes care of converting the Python arguments to Praat and returns the objects selected at the end of the script (similar to `call`, cfr. Section 2.3).

After running the Praat script, we get a `TextGrid` object and query it with `praat.call` – as we currently do not yet have a Python interface to `TextGrid` in Parselmouth – to get the estimated syllable centre timings. Note that we only use Parselmouth for running the script and getting its results; the rest of the example code uses standard Python code and libraries to loop over all files of the corpus and store the results in one shared *pandas* `DataFrame`. We refer to the supplementary material for a version of the script with comments and more details on this part of the code.

After getting the intervals between syllables for all audio files in the corpus, we can run the desired statistical analysis on the data, as illustrated in Listing 6. We fit two different implementations of mixed-effects linear models, one being a maximum likelihood estimation and the other following a Bayesian approach. Since we have extracted the syllable nuclei and intervals in Listing 5 already, Praat and Parselmouth are actually not involved in this second part of the example. Consequently, it would be possible to write and run the first part of the code as Praat script, write the data to file, and load that file when running the statistical analysis. Rather than claiming our approach using Parselmouth to be better or easier, we merely want to demonstrate the possibility of a complete Python workflow, as motivated before (cfr. Section 1.1).

Once again, our complete example mainly aims to put forward how a Parselmouth user could integrate already existing Praat scripts into a new Python script or project (i.e., through the `call`, `run`, and `run_file` functions in the `praat` submodule). Moreover,

[39] `http://www.statsmodels.org/`
[40] `https://github.com/bambinos/bambi`
[41] `http://docs.pymc.io/`
[42] `http://www.fon.hum.uva.nl/praat/manual/Scripting_6_1__Arguments_to_the_script.html`

```
import parselmouth
from parselmouth.praat import call, run_file

import numpy as np
import pandas as pd

def extract_syllable_intervals(file_name):
        print("Extracting syllable intervals from '{}'...".format(file_name))

        # Use Praat script to extract syllables
        objects = run_file('syllable_nuclei.praat', -25, 2, 0.3, file_name)
        textgrid = objects[1]
        n = call(textgrid, "Get number of points", 1)
        syllable_nuclei = [call(textgrid, "Get time of point", 1, i + 1)
                           for i in range(n)]
        # Use NumPy to calculate intervals between the syllable nuclei
        syllable_intervals = np.diff(syllable_nuclei)
        return syllable_intervals


def syllable_intervals_data(row):
        # Get file name of corpus audio file
        file_name_format = "corpus/dialectaccent_vol_01_{:02}{}_64kb.mp3"
        file_name = file_name_format.format(row['audio_id'], row['speaker'])

        # Extract syllables and intervals with previously defined function
        intervals = extract_syllable_intervals(file_name)

        # Return a new data frame with a row for each extracted interval
        return pd.DataFrame({'speaker': row['speaker'],
                             'native': row['native'],
                             'interval': intervals})


corpus = pd.read_csv("corpus/corpus.csv")
# Concatenate all data from the corpus into one big pandas DataFrame
data = pd.concat([syllable_intervals_data(row) for _, row in corpus.iterrows()])
```

Listing 5: Already existing Praat scripts can be run through the `parselmouth.praat.run` and `parselmouth.praat.run_file` functions to interface with the use of Parselmouth objects and standard Python variables. Usage of Parselmouth functionality is highlighted in red; a version with detailed comments can be found in the supplementary material.

we think the versatile range of less-common statistical methods available outside Praat are a good illustration of why the integration of Praat and Python can be useful. We could of course combine this example with visualization capabilities similar to the ones laid out in the previous examples (cfr. Section 2.1) to achieve a large portion of the speech data analyst's workflow in one language.

More generally, while few, if any, of the practices we have laid out in these examples so far are technically *impossible* to achieve using Praat alone (with the appropriate level of expertise), Parselmouth improves efficiency, by facilitating integration of the four major strands of speech data analysis: reading and writing audio; acoustic analysis; statistical analysis; and visualization. Furthermore, and maybe even more crucially in our view,

```
# Maximum likelihood (ML/REML) estimation of mixed-effects linear model
import statsmodels.formula.api as smf

model = smf.mixedlm('interval ~ native', data, groups=data['speaker'])
results = model.fit()
print(results.summary())


# Bayesian estimation of mixed-effects linear model
import bambi

model = bambi.Model(data)
results = model.fit('interval ~ native', random=['1|speaker'])
print(results.summary())
```

Listing 6: Once the necessary data is extracted from the corpus (cfr. Listing 5), it can directly be analysed using specialised statistical libraries in Python. A version with detailed comments can be found in the supplementary material.

this kind of access to existing scripts with Parselmouth can help to eventually expand the range of users who are able to make use of Praat's functionality.

2.5. Integration into Experimental Design

Contemporary experimental procedures increasingly require sophisticated computational workflows. In the speech-related sciences, it has traditionally been difficult to build automated acoustic data analysis into experimental procedures because trial structure design and implementation is typically not programmed in Praat (though we note that experimental design is actually possible in Praat[43]), but in specialised experimental software packages that do not include acoustic data manipulation and analysis tools. Parselmouth can help solve this problem through integration with widely-used Python-based experimental software such as PsychoPy (Peirce, 2007, 2009). As a simple example of this capability, this section shows how Parselmouth can be built into the trial loop of a standard adaptive staircase design experiment that is part of PsychoPy. Such an experimental design could for example be used determine just-noticeable differences, or as pre-experimental routine to test a participant's hearing or adjust to the level of background noise. Although we focus on this particular experimental procedure, we emphasise that this example is part of a broader class of adaptive, algorithmic experimental designs that are expected to become increasingly important in experimental research (Suchow & Griffiths, 2016): Parselmouth, providing the possibility of bringing both acoustic analysis and synthesis into the experimental loop, helps put these kinds of experiments within easier reach of speech scientists using Python.

Although programming an interactive experiment is also possible with the Praat scripting language, as interaction with the participant is supported through the Praat demo window[44] and anything else can be programmed from scratch, we see a few advantages to using an experimental package like PsychoPy: firstly, PsychoPy is an established, widely-used software package, developed around the central idea of running

[43] E.g., `http://www.fon.hum.uva.nl/praat/manual/ExperimentMFC.html`

[44] `http://www.fon.hum.uva.nl/praat/manual/Demo_window.html`

"neuroscience, psychology and psychophysics experiments"[45]. Secondly, PsychoPy already contains many built-in experimental features and has a graphical user interface that allows for quickly setting up an experiment without writing any code. And finally, a researcher with more advanced needs *can* escape this purely graphical interface and add custom components and code to the experiment, while still taking advantage of the functionality that is already available in PsychoPy and focusing on this custom functionality.

An introduction to staircase experimental design (e.g., Kaernbach, 2001) and to the details of PsychoPy's implementation of this class of experimental designs, can be found in a PsychoPy tutorial on measuring just-noticeable-differences, *"Measuring a JND using a staircase procedure"*[46]. Generally speaking, PsychoPy can be used in two different ways: it is possible to write a Python script that imports and uses the `psychopy` module, or one can use the graphical *Builder* interface, as shown in Figure 3. In the first case, the user can just import and use the `parselmouth` library alongside `psychopy`, but also when using the graphical user interface, Praat's algorithms and potential can be accessed through Parselmouth. Crucially, through the addition of *Code components* the PsychoPy Builder allows blocks of Python code to be inserted into an experimental design such that these code blocks are executed on initialisation, during, or on completion of one of the experiment's routines (cfr. Figure 3). This is how arbitrary Praat functionality can be inserted into the experimental procedure to generate and update custom acoustic stimuli on the fly.

As an illustration of this principle and its simplicity, we have implemented a staircase design experiment. It was engineered to converge on the lowest signal-to-noise ratio at which participants can correctly classify a Gaussian white-noise corrupted speech segment in which the speaker says either *'bet'* or *'bat'*, loosely based on an experiment by de Boer (2012). Such a design requires that at the start of each trial, white noise is added to an audio stimulus, to a degree that is determined by the participant's response in the previous trial. Moreover, the resulting stimulus must then be rescaled to have a constant mean intensity. These computations can be handled easily using Parselmouth, especially when one is already familiar with how to do this in Praat. Listing 7 shows the code inserted into the PsychoPy trial. The full PsychoPy builder project as well as the generated code to run the experiment are provided as supplementary material, together with the two associated audio files. This example, like the ones before, can serve as a starting point for building one's own Python scripts and experiments with Parselmouth.

While PsychoPy provides a convenient framework to set up an experiment in Python, Parselmouth also allows for Praat functionality to be accessed from a generic Python model or experiment. For example, Rasilo & Räsänen (2017) describe an online model of language acquisition where a "learning virtual infant" interacts with a human caregiver in an experiment. The model of the babbling infant combines different aspects of learning, including an articulatory model, the clustering and categorisation of the different babbled utterances, and an algorithm to learn associations between these utterances and the caregiver's responses. Consequently, most of the implementation of this computational model would be difficult in the Praat scripting language, as the model might benefit from using computational libraries. However, the model also involves the extraction of

[45] `http://www.psychopy.org/`

[46] `http://www.psychopy.org/coder/tutorial2.html`

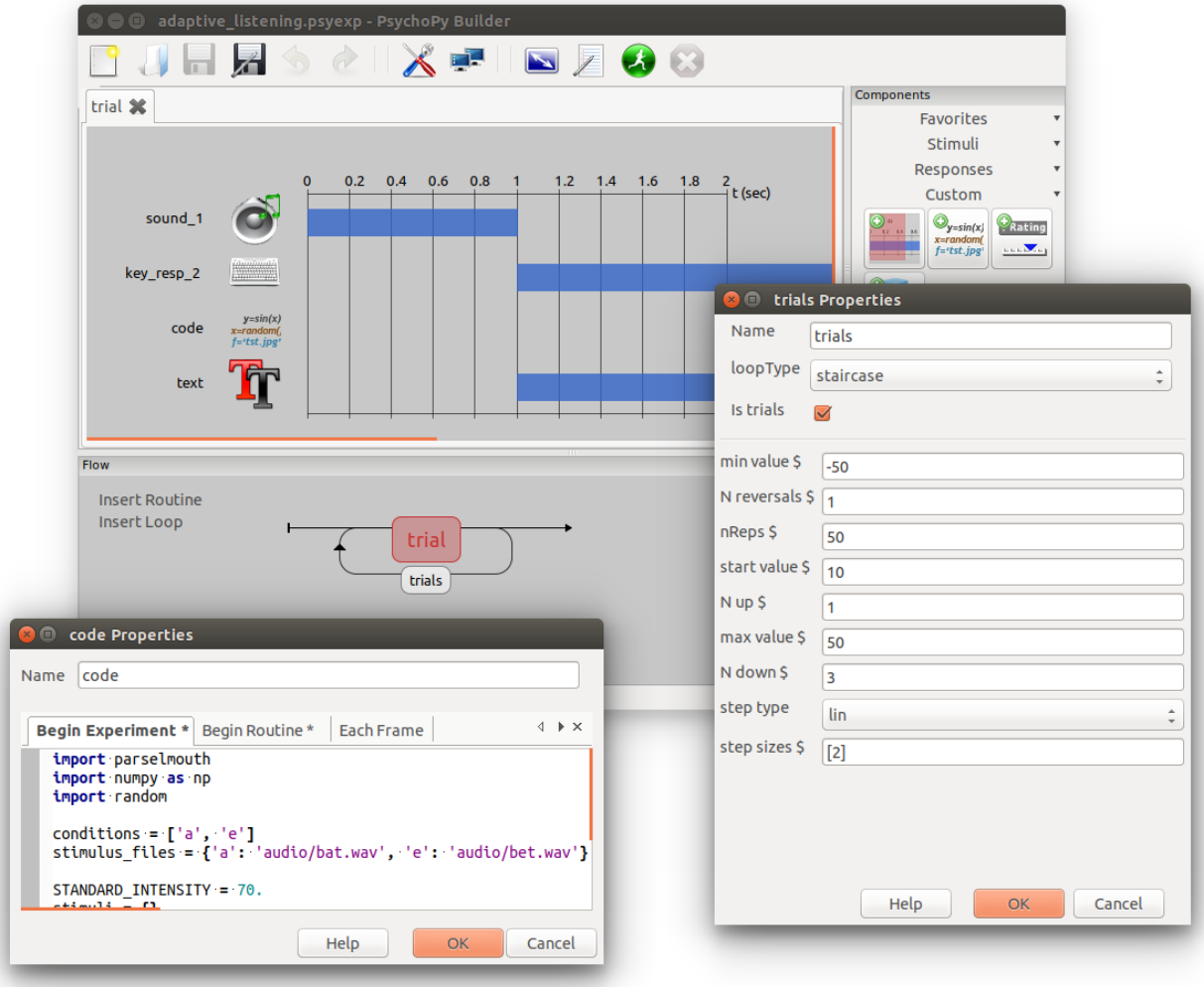

Figure 3: A screenshot of the *PsychoPy Builder* showing how the built-in 'staircase' *loopType* and the custom Python code using Parselmouth (cfr. Listing 7) fit into the overall PsychoPy experiment.

formant frequencies when converting a speech utterance into an acoustic representation. Since this is only a small aspect of the full model, this is most probably not enough to encourage a researcher to switch to using Praat, while the model could benefit from using the established Praat functionality for formant analysis. More generally, this is why we believe that also for a broader range of computational models and interactive experiments, acoustic feature extraction and articulatory speech synthesis could potentially be done using the Praat features whereas the rest of the framework can then be implemented independently, outside Praat. The use of Python as glue language and the flexibility and modularity of the Python ecosystem (as described in Section 1.1) would

```
# -- Begin experiment --

import parselmouth
import numpy as np
import random

conditions = ['a', 'e']
stimulus_files = {'a': 'audio/bat.wav', 'e': 'audio/bet.wav'}

STANDARD_INTENSITY = 70.
stimuli = {}
for condition in conditions:
    stimulus = parselmouth.Sound(stimulus_files[condition])
    stimulus.scale_intensity(STANDARD_INTENSITY)
    stimuli[condition] = stimulus
# -- Begin Routine --

random_condition = random.choice(conditions)
random_stimulus = stimuli[random_condition]

noise_samples = np.random.normal(size=random_stimulus.n_samples)
noisy_stimulus = parselmouth.Sound(noise_samples,
                    sampling_frequency=random_stimulus.sampling_frequency)
noisy_stimulus.scale_intensity(STANDARD_INTENSITY - level)
noisy_stimulus.values += random_stimulus.values
noisy_stimulus.scale_intensity(STANDARD_INTENSITY)

# 'filename' variable is set by PsychoPy and contains base file name
# of saved log/output files, so we'll use that to save our custom stimuli
stimulus_file_name = filename + '_stimulus_' + str(trials.thisTrialN) + '.wav'
noisy_stimulus.resample(44100).save(stimulus_file_name, "WAV")
sound_1.setSound(stimulus_file_name)
# -- End routine --

trials.addResponse(key_resp_2.keys == random_condition)
```

Listing 7: Code snippets accessing Parselmouth functionality inserted into a PsychoPy experiment through a Code Component, respectively in the *Before Experiment*, *Begin Routine*, and *End Routine* section. Usage of Parselmouth functionality is highlighted in red; a version with detailed comments can be found in the supplementary material.

then allow Praat to be used in such a context, through Parselmouth.

These five presented usage examples are meant so show the variety of situations in which a researcher might elect to use Parselmouth, but rather than demonstrating the functionality per se, we wanted to show how Parselmouth can facilitate the combination of the specialised functionality of Praat with the wide range of available software packages and computational environments. After all, these simplified examples of less than 50 lines of Python code are only scratching the surface of the diversity of tasks phoneticians, linguists, and other scientists face. With Parselmouth, we hope to have provided researchers with a tool to enable an easier implementation of more advanced scientific models and experiments. Just to illustrate with a final hypothetical example,

one could even go as far as using a Python web server framework (for instance, Flask[47] or Django[48]) to provide a web service that involves acoustic processing. Such a web service could then be used in the context of a crowd-sourced experiment (where users only run a simple JavaScript program locally), potentially even in combination with Amazon Mechanical Turk[49] which today is sometimes used to recruit participants in large-scale online experiments (see, for example, the Dallinger experiment automation framework[50]).

3. Conclusion

We hope to have illustrated how Parselmouth can be useful as a Python interface to Praat. Though our usage examples focused on visualisation, data file manipulation, audio manipulation, statistical libraries, and integration into a PsychoPy experiment, we envisage an unbounded range of practical applications for the package (e.g., various machine learning libraries and deep learning frameworks that are gaining popularity, as mentioned in Section 1.1). The aim of Parselmouth is to link Praat and Python; Parselmouth gives a Praat user access to the wide variety of scientific and utility packages available for Python, but also hands over control of Praat's functionality to a Python user in a way that naturally generalises the user's experience of programming in Python. We hope that such a package can significantly broaden the user base for Praat's technology.

We wish to end by stressing again that Parselmouth *relies upon* Praat, rather than replacing it: usage of Parselmouth implies usage of Praat, with its expansive collection of code and sophisticated algorithms.

3.1. Future Directions & Development

Parselmouth is under active development. We welcome contribution from others in the community: the project requires not just code, but user feedback, bug reports, feature requests, usage examples, tutorials, and documentation. We believe there is a strong demand for sophisticated phonetic data analysis tools in Python, and this demand can only be fulfilled through community driven efforts. Anybody interested in contributing or providing feedback or requests can email the first author of this paper, or visit the dedicated Parselmouth chat room established on Gitter[51].

Acknowledgements

YJ would like to thank Robin Jadoul for repeatedly answering questions like "How would you prefer to write such-and-such in Python?", and for being my Virgil in the DLL Inferno, and to thank his co-authors, Piera Filippi, and Andrea Ravignani for their lasting enthusiasm and encouragements during the development of Parselmouth and for proof-reading the manuscript, as well as Katie Mudd, Marnix Van Soom, and Marianne de Heer Kloots for their feedback, encouragement, and proliferation of Parselpropaganda.

[47] `http://flask.pocoo.org/`
[48] `https://www.djangoproject.com/`
[49] `https://www.mturk.com/`
[50] `https://github.com/Dallinger/Dallinger`
[51] `https://gitter.im/PraatParselmouth/Lobby`

Funding: This project was supported by a PhD Fellowship (Aspirant) of the Research Foundation Flanders - Vlaanderen (FWO) to YJ, and European Research Council grant 283435 ABACUS to BdB.

References

Albin, A. L. (2014). PraatR: an architecture for controlling the phonetics software "Praat" with the R programming language. *The Journal of the Acoustical Society of America*, *135*, 2198–2199.

de Boer, B. (2012). Loss of air sacs improved hominin speech abilities. *Journal of human evolution*, *62*, 1–6.

Boersma, P. (2001). PRAAT, a system for doing phonetics by computer. *Glot International*, *5*, 341–345.

Boersma, P., & Weenink, D. (2018). Praat: doing phonetics by computer [Computer program]. Version 6.0.40, retrieved 11 May 2018 from `http://www.praat.org/`.

Bořil, T., & Skarnitzl, R. (2016). Tools rPraat and mPraat. In P. Sojka, A. Horák, I. Kopeček, & K. Pala (Eds.), *International Conference on Text, Speech, and Dialogue* (pp. 367–374). Springer International Publishing.

De Jong, N. H., & Wempe, T. (2009). Praat script to detect syllable nuclei and measure speech rate automatically. *Behavior research methods*, *41*, 385–390.

Filippi, P., Gingras, B., & Fitch, W. (2014). Pitch enhancement facilitates word learning across visual contexts. *Frontiers in psychology*, *5*, 1468.

Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. *Computing In Science & Engineering*, *9*, 90–95.

International Phonetic Association (1999). *Handbook of the International Phonetic Association: A guide to the use of the International Phonetic Alphabet*. Cambridge University Press.

Jadoul, Y., Ravignani, A., Thompson, B., Filippi, P., & de Boer, B. (2016). Seeking temporal predictability in speech: comparing statistical approaches on 18 world languages. *Frontiers in human neuroscience*, *10*, 586.

Jakob, W., Rhinelander, J., & Moldovan, D. (2017). pybind11 – Seamless operability between C++11 and Python. `https://github.com/pybind/pybind11`.

Kaernbach, C. (2001). Adaptive threshold estimation with unforced-choice tasks. *Attention, Perception, & Psychophysics*, *63*, 1377–1388.

McKinney, W. (2010). Data structures for statistical computing in Python. In *Proceedings of the 9th Python in Science Conference* (pp. 51–56). SciPy Austin, TX volume 445.

Peirce, J. W. (2007). PsychoPy – psychophysics software in python. *Journal of neuroscience methods*, *162*, 8–13.

Peirce, J. W. (2009). Generating stimuli for neuroscience using PsychoPy. *Frontiers in neuroinformatics*, *2*, 10.

Rasilo, H., & Räsänen, O. (2017). An online model for vowel imitation learning. *Speech Communication*, *86*, 1–23.

Ravignani, A. (2018). Spontaneous rhythms in a harbor seal pup calls. *BMC research notes*, *11*, 3.

Suchow, J. W., & Griffiths, T. L. (2016). Rethinking experiment design as algorithm design. In *CrowdML – NIPS '16 Workshop on Crowdsourcing and Machine Learning*.

Walt, S. v. d., Colbert, S. C., & Varoquaux, G. (2011). The NumPy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, *13*, 22–30.