forum acusticum 2023

INTRODUCING VOCALPY: A CORE PYTHON PACKAGE FOR RESEARCHERS STUDYING ANIMAL ACOUSTIC COMMUNICATION

David Nicholson[1*]
[1] Independent Researcher, United States of America

ABSTRACT

The study of animal acoustic communication requires true interdisciplinary collaboration, big team science, and cutting edge computational methods. To those ends, more and more groups have begun to share their code. However, this code is often written to answer very specific research questions, and tailored to lab-specific data formats. As a result, it is not always easy to read and reuse, and there is significant duplication of effort. Here I introduce a Python package, VocalPy, created to address these issues. VocalPy has two main goals: (1) make code more readable across research groups, and (2) facilitate collaboration between scientists-coders writing analysis code and research software engineers developing libraries and applications. To achieve these goals, VocalPy provides a set of software abstractions for acoustic communication research. These abstractions encapsulate common data types, such as audio, spectrograms, acoustic features, and annotations. Additional abstractions represent typical steps in workflows, e.g., segmenting audio into sequences of units, computing spectrograms, and extracting features. I demonstrate by example how these abstractions in VocalPy enable scientist-coders to write more readable, idiomatic analysis code, that is more easily translated to an application run at scale.

Keywords: bioacoustics, Python, animal communication, acoustic communication, speech

*__Corresponding author__: nicholdav@gmail.com.

1. INTRODUCTION

The study of how animals communicate with sound gets at questions that are central to what it means to be human. How did language evolve, and how does it relate to the ability of vocal learning in other animals [1, 2]? Answering these questions requires collaboration across disciplines, big team science, and cutting edge computational methods. Many authors have called for large scale collaboration across disciplines to investigate language (as in [1]), and have highlighted the interdisciplinary nature of acoustic communication research more generally (as in [2]). Concurrently, the many disciplines studying acoustic communication are becoming ever more computational, and it has become clear that cutting edge computational methods will play a key role in this research area. To see this, one need look no further than the widespread proliferation of deep learning models, as applied to the neuroethology of vocal communication [3] and to bioacoustics more generally [4].

Broadly speaking, workers in this area have applied computational methods and shared the results in one of two ways, both having their own strengths and weaknesses. The first way is through graphical user interface (GUI) software, and the second is through imperative scripts for analysis. The strength of GUI software is that it allows researchers to carry out sophisticated analyses without programming knowledge. A significant drawback is that GUIs (usually) do not capture all steps of analysis, at least not in a manner that makes it easy for anyone to replicate. Partly in reaction to this, more and more acoustic communication researchers run their analyses with scripts. These researchers are also sharing this code with their data, to improve the replicability of their results. In other words, they are adopting the open science practices pioneered by other fields that rely heavily

on computational methods.

However, acoustic communication researchers sharing code across disciplines and research groups results in its own set of issues, especially when that code uses rapidly evolving computational methods. First, code associated with a publication is often written to answer very specific research questions. Second, the same code is also tailored to very specific data formats, which vary widely across groups. In particular, in the Python programming language, there is no core package for acoustic communication researchers. Instead, scientist-coders tend to write verbose scripts with multiple related variables passed from function to function, even when these variables have a natural association that could be encapsulated with a data type. I provide an example code snippet to illustrate this in Listing 1 below. As a result of all these factors, it is not always easy to read and reuse shared code. Furthermore, because each group writes code to deal with low-level details, there is massive duplication of effort.

To address these issues, and to explore what a core Python package for acoustic communication research might look like, here I introduce VocalPy (`https://github.com/vocalpy/vocalpy`). VocalPy addresses these issues with an approach loosely inspired by domain-driven design [5]. Thus, the architecture of VocalPy is based on a domain model of common workflows in acoustic communication research. In the terminology of domain-driven design, this domain model is meant to capture the essential entities, services, and relationships in these workflows. Entities are uniquely-identifiable data we need to track over the lifetime of a workflow: audio signals, array data such as spectrograms and extracted acoustic features, and annotations such as those that researchers produce with GUI applications. Services can convert one form of acoustic communication data to another: a spectrogram is made from audio, features are extracted from a spectrogram or annotation file, and a set of files is persisted to a database to represent a dataset. To illustrate the entities and services provided by VocalPy, a schematized version of two common workflows in acoustic communication research are shown in Fig. 1. These will be presented further as code listings below. One reason to base the design of VocalPy on a domain model is to express workflows in a way that minimizes the distance between an analysis in an imperative script, as might be written by a domain expert with less coding knowledge, and similar functionality provided by an application, as developed by a research software engineer (which would be a true service as the term is used in domain-driven design). I want to make my debt to domain-driven design clear here, but for clarity of presentation below I dispense with terms like ”entities” and ”services” in favor of more familiar terms like data types and classes.

The rest of the paper is structured as follows: first I provide a code listing highlighting some issues with Python code for acoustic communication research as it is often written now. Next I introduce the data types and classes in VocalPy meant to help make code more readable and idiomatic across research groups. Finally I provide an example of a common workflow as it would be written with VocalPy: segmenting audio into sequences of units for further analysis [6].

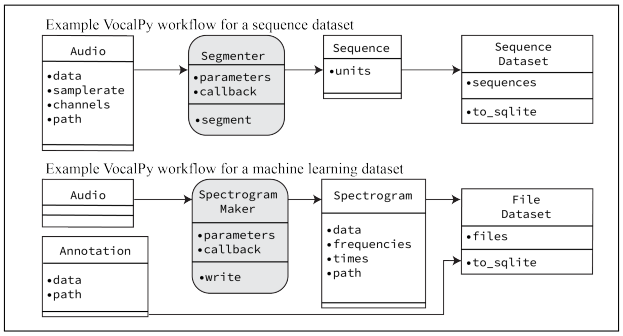

Figure 1. Schematic of two common workflows in research on acoustic communication, that illustrate core data types and other classes provided by VocalPy. Classes are shown as a simple Unified Markup Language (UML) diagram, rectangles divided in three with the class' name at the top, its attributes in the middle, and its methods in the bottom. Only attributes and methods that are key for discussion here are shown.

2. DESIGN OF VOCALPY

2.1 Comparison to code written without VocalPy

I began by further motivating the need for VocalPy with an example listing. Please know I am not suggesting there is anything wrong with this code per se; I have written code like this in myself in the past. My goal is to illustrate how such code can be easier to both write and read.

Because of space considerations, I cannot provide a lengthy example that gives the full effect of reading an entire set of scripts for a project. But we can notice several things that are common in such scripts in Listing 1. First

```python
from scipy.signal import spectrogram
import soundfile

def spect(data, fs, fft=1024, window='Hann'):
    f, t, s = spectrogram(data, fs, fft=fft, window=window)
    return f, t, s

data_bird1, fs_bird1 = soundfile.read('./path/to/bird1.wav')
f_bird1, t_bird1, s_bird1  = spect(data_bird1, fs_bird1)
data_bird2, fs_bird2 = soundfile.read('./path/to/bird2.wav')
f_bird2, t_bird2, s_bird2  = spect(data_bird2, fs_bird2)

# definitions of functions below are not shown in snippet
ftrs_bird1 = extract_features(s_bird1, t_bird1, f_bird1)
ftrs_bird2 = extract_features(s_bird2, t_bird2, f_bird2)
rejected_h0, pval = stats_helper(ftrs_bird1, ftrs_bird2)
```

Listing 1: Toy example of a typical script for acoustic communication research, written using standard scientific Python, without VocalPy.

we notice a helper function to generate a spectrogram, that has two required arguments: the audio signal and its sampling rate. Both are required, but without a data type that encapsulates them, we must pass them in separately. Next, notice that the helper function also has several default arguments. Often these default values hidden in such helper functions can turn out to be key parameters in a scientist-coder's analysis. As a reader, we may only be able to determine this by combining clues across multiple scripts. I show how VocalPy avoids this in Listing 3. Also note that the helper function returns multiple arrays: the matrix representing the spectrogram itself, as well as the vectors representing the frequencies and times in the spectrogram. Often we need all three of these for certain analyses, such as extracting an acoustic parameter within a specific time and frequency range. Again, because there is no data type to represent spectrograms, we are required to pass multiple related variables around to our functions. Finally, notice that these variables can multiply as we try to represent multiple conditions in our code. In this case I use suffixes (`_bird1`, `_bird2`) to distinguish the same data types from two different birds. This pattern is common in imperative analysis code written by someone familiar with MatLab and numpy, but less accustomed to leveraging native Python types or a tidy data approach that might represent conditions with a categorical variable. (The author was such a someone, once.)

2.2 VocalPy Data Types

Next I rewrite Listing 1 using VocalPy, to introduce its data types.

We can observe several differences when compared with Listing 1. First notice that in Listing 2 we represent

```python
import vocalpy as voc
from scipy.signal import spectrogram

def spect(audio, fft=1024, window='Hann'):
    f, t, s = spectrogram(audio.data, audio.samplerate,
                          fft=fft, window=window)
    return voc.Spectrogram(data=s, frequencies=f, times=t)

ftrs = {}
for bird in ('bird1', 'bird2'):
    audio = voc.Audio.read(f'./path/to/{bird}.wav')
    spect = spect(audio)
    ftrs[bird] = extract_features(spect)

rejected_h0, pval = stats_helper(ftrs['bird1'], ftrs['bird2'])
```

Listing 2: Listing 1 rewritten with VocalPy data types.

audio with the `vocalpy.Audio` data type, loading a file with its `read` method. We do still have a helper function that computes spectrograms, whose default parameters could hide key parameters in our analysis; in the next section I show how to avoid this potential drawback using the `SpectrogramMaker` class built into VocalPy. Here the helper function lets us see that, instead of passing multiple arrays around, we can instead pass in a single data type, `vocalpy.Audio`, and return a single data type, `vocalpy.Spectrogram`. Both of these data types encapsulate related attributes in a single class.

2.3 VocalPy classes for workflows and datasets

Finally I introduce two more types of classes in VocalPy. The first represents steps in workflows. The second represents datasets, and captures metadata about how the datasets were created. Listing 3 shows a session in the Python REPL, to demonstrate how VocalPy's design is meant to make it easy for a scientist-coder to work interactively. The commands in this session constitute the initial steps of any workflow for analyzing sequences of units [6] (simplified for presentation), as depicted schematically in the top row of Figure 1.

I highlight some important features of the listing. First notice that here we explicitly declare the parameters we use to segment audio into units, as a Python dictionary. We pass these parameters to a class that represents the process of segmenting, `vocalpy.Segmenter`. To segment audio, the parameters are passed to a callback function. This function is passed in as an argument, in this case `evfuncs.segment_song`. Please note some key aspects of this design: it encourages us to clearly state what parameters we use, to avoid hiding them in a helper function. It also captures the function we use to segment,

```
>>> import evfuncs
>>> import vocalpy as voc
>>> data_dir = 'gy6or6/032312/'
>>> cbin_paths = voc.paths.from_dir(data_dir, 'cbin')
>>> audios = [voc.Audio.read(cbin_path)
...     for cbin_path in cbin_paths]
>>> segment_params = {'threshold': 1500, 'min_syl_dur': 0.01,
...     'min_silent_dur': 0.006}
>>> segmenter = voc.Segmenter(
...     callback=evfuncs.segment_song,
...     segment_params=segment_params)
>>> seqs = segmenter.segment(audios, parallel=True)
>>> seq_dataset = voc.dataset.SequenceDataset(sequences=seqs)
>>> seq_dataset.to_sqlite(db_name='gy6or6-032312.db',
...     replace=True)
>>> print(seq_dataset)
SequenceDataset(sequences=[Sequence(units=
[Unit(onset=2.18934375, offset=2.21, label='-',
audio=None, spectrogram=None),
Unit(onset=2.346125, offset=2.373125, label='-',
audio=None, spectrogram=None),
# rest of output omitted
>>> # test that we can load the dataset and it compares equal
>>> loaded = voc.dataset.SequenceDataset.from_sqlite(
...     db_name='gy6or6-032312.db')
>>> loaded == seq_dataset
True
```

Listing 3: Use of VocalPy in the Python REPL to build a dataset of sequences

the callback. Additionally, the callback-based design affords research groups the ability to re-use their existing code. Once the `vocalpy.Segmenter` class is instantiated, we call its `segment` method that returns a Python list of `vocalpy.Sequence` instances, one for each `vocalpy.Audio` instance we pass in. Each sequence has as attributes its source audio, as well as the segmenting parameters and callback used to segment. This enables us to create a `vocalpy.dataset.SequenceDataset` from the `vocalpy.Sequence` instances that automatically traces the provenance of our data: which audio gave us which sequence, and how was that audio segmented. Finally we call the dataset class' method `to_sqlite`, to persist the dataset to disk in a single-file database. In this way, a scientist-coder can flexibly build a dataset and save it to a shareable file, without needing to install or use a database directly. We choose to default to SQLite for several reasons, the two most important of which are that it is built into Python, and it is one of four storage formats recommended for datasets by the United States Library of Congress (`https://www.sqlite.org/locrsf.html`).

3. DISCUSSION

Here I introduced VocalPy. Its design represents what I have argued is needed for a core Python package for acoustic communication. Through example listings I presented the core data types, and demonstrated how the built-in classes support common workflows such as analysis of acoustic sequences described in [6]. It is my hope that this introduction will further motivate all of us in this research area to create the community-developed software that we need to collaborate and communicate across research groups and disciplines.

4. REFERENCES

[1] M. D. Hauser, N. Chomsky, and W. T. Fitch, "The Faculty of Language: What Is It, Who Has It, and How Did It Evolve?," *Science*, vol. 298, pp. 1569–1579, Nov. 2002.

[2] M. Wirthlin, E. F. Chang, M. Knörnschild, L. A. Krubitzer, C. V. Mello, C. T. Miller, A. R. Pfenning, S. C. Vernes, O. Tchernichovski, and M. M. Yartsev, "A Modular Approach to Vocal Learning: Disentangling the Diversity of a Complex Behavioral Trait," *Neuron*, vol. 104, pp. 87–99, Oct. 2019.

[3] T. Sainburg and T. Q. Gentner, "Toward a Computational Neuroethology of Vocal Communication: From Bioacoustics to Neurophysiology, Emerging Tools and Future Directions," *Frontiers in Behavioral Neuroscience*, vol. 15, p. 811737, Dec. 2021.

[4] D. Stowell, "Computational bioacoustics with deep learning: A review and roadmap," p. 46, 2022.

[5] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2004.

[6] A. Kershenbaum, D. T. Blumstein, M. A. Roch, Ç. Akçay, G. Backus, M. A. Bee, K. Bohn, Y. Cao, G. Carter, C. Cäsar, M. Coen, S. L. DeRuiter, L. Doyle, S. Edelman, R. Ferrer-i-Cancho, T. M. Freeberg, E. C. Garland, M. Gustison, H. E. Harley, C. Huetz, M. Hughes, J. Hyland Bruno, A. Ilany, D. Z. Jin, M. Johnson, C. Ju, J. Karnowski, B. Lohr, M. B. Manser, B. McCowan, E. Mercado, P. M. Narins, A. Piel, M. Rice, R. Salmi, K. Sasahara, L. Sayigh, Y. Shiu, C. Taylor, E. E. Vallejo, S. Waller, and V. Zamora-Gutierrez, "Acoustic sequences in non-human animals: A tutorial review and prospectus: Acoustic sequences in animals," *Biological Reviews*, vol. 91, pp. 13–52, Feb. 2016.