GhostiPy: an efficient signal processing and spectral analysis toolbox for large data

<https://doi.org/10.1523/ENEURO.0202-21.2021>

Cite as: eNeuro 2021; 10.1523/ENEURO.0202-21.2021

Received: 5 May 2021

Revised: 3 August 2021

Accepted: 31 August 2021

This Early Release article has been peer-reviewed and accepted, but has not been through the composition and copyediting processes. The final version may differ slightly in style or formatting and will contain links to any extended data.

Alerts: Sign up at www.eneuro.org/alerts to receive customized email alerts when the fully formatted version of this article is published.

Copyright © 2021 Chu and Kemere

This is an open-access article distributed under the terms of the Creative Commons Attribution 4.0 International license, which permits unrestricted use, distribution and reproduction in any medium provided that the original work is properly attributed.

- 1 1. **Manuscript title:** GhostiPy: an efficient signal processing and spectral analysis toolbox for large
- 2 data
- 3 2. **Abbreviated title:** GhostiPy: efficient software for large data
- 4 3. **Author affiliations:** Joshua P. Chu, Department of Electrical and Computer Engineering, Rice
- 5 University, Houston, TX; Caleb T. Kemere, Department of Electrical and Computer Engineering,
- 6 Rice University, Houston, TX
- 7 4. **Author contributions:** JC developed the software, analyzed data, wrote the paper. CK wrote the
- 8 paper.
- 9 5. **Correspondence email:** caleb.kemere@rice.edu
- 10 6. **Number of figures:** 7
- 11 7. **Number of tables:** 1
- 12 8. **Number of multimedia:** 0
- 13 9. **Number of words for abstract:** 143
- 14 10. **Number of words for significance statement:** 103
- 15 11. **Number of words for introduction:** 277
- 16 12. **Number of words for discussion:** 366
- 17 13. **Acknowledgements:** We thank Shayok Dutta (Rice University), Antonio Fernandez-Ruiz (Cornell
- 18 University) and Josh Siegle (Allen Institute) for sharing data used in example analyses.
- 19 14. **Conflict of interest:** Authors report no conflict of interest
- 20 15. **Funding sources:** The development of GhostiPy was supported by the National Science
- 21 Foundation (NSF CBET1351692) and the National Institute of Neurological Diseases and Strokes
- 22 (R01NS115233).

GhostiPy: an efficient signal processing and spectral analysis toolbox for large data

Abstract

Recent technological advances have enabled neural recordings consisting of hundreds to thousands of channels. As the pace of these developments continues to grow rapidly, it is imperative to have fast, flexible tools supporting the analysis of neural data gathered by such large scale modalities. Here we introduce **ghostipy** (**g**eneral **h**ub **o**f **s**pectral **t**echniques **i**n **P**ython), a Python open source software toolbox implementing various signal processing and spectral analyses including optimal digital filters and time-frequency transforms. **ghostipy** prioritizes performance and efficiency by using parallelized, blocked algorithms. As a result, it is able to outperform commercial software in both time and space complexity for high channel count data and can handle out-of-core computation in a user-friendly manner. Overall, our software suite reduces frequently encountered bottlenecks in the experimental pipeline, and we believe this toolset will enhance both the portability and scalability of neural data analysis.

Significance Statement

Due to technological innovation the size of neural recordings has increased dramatically, but downstream analysis code is often not optimized to handle such large scales of data efficiently. Here we have developed GhostiPy, an open source Python package prioritizing

performance and efficiency for large data in the context of typical spectral analysis and signal processing algorithms. Users can control hardware resource consumption (such as system memory) by setting the level of parallelization and enabling out-of-core processing. Thus algorithms can be run on a variety of hardware, from laptops to dedicated compute servers. Overall, GhostiPy improves experimental throughput by increasing the portability of analyses.

Introduction

Advancements in neural recording technologies have enabled the collection of large data in both space (high density/channel count) and time (continuous recordings). During subsequent analysis, the scale of the data induces certain challenges which may manifest as the following scenarios: (1) analysis code takes a long time to complete (high time complexity), (2) code is unable to complete due to insufficient memory on the hardware (high spatial complexity). Moreover the scientist may have difficulty finding existing tools that address both (1) and (2) and that implement the desired analyses.

Although a potential remedy is to simply upgrade the hardware, it is not an acceptable solution for scientists desiring portability, an important component that improves reproducibility and replicability. In more portable systems hardware resources may be limited (e.g. using a laptop at the airport). We thus took an alternate approach by efficiently implementing analyses that would trivially scale for different hardware configurations. Our solution is **ghostipy** (general hub of spectral techniques in Python), a free and open source Python toolbox that attempts to optimize both time and space complexity in the context of spectral analyses. Methods include linear filtering, signal envelope extraction, and spectrogram estimation, according to best practices. **ghostipy** is designed for general purpose usage; while well suited for high density continuous neural data, it works with any arbitrary array-like data object.

In this paper we first describe `ghostipy`'s software design principles to increase efficiency. We then elaborate on featured methods along with code samples illustrating user friendliness of the software. Finally we benchmark our software against a comparable implementation, and we discuss strategies for working under an out-of-core (when data cannot fit into system memory) processing context.

Materials and methods

An overview of implemented methods can be found in Table 1. Excluding out-of-core support, it is possible to use multiple different packages (Bokil et al. 2010; Gramfort et al. 2013; Lee et al. 2019; Oostenveld et al. 2011; OverLordGoldDragon 2020; Tadel et al. 2019; Virtanen et al. 2020; Yegenoglu et al. 2015) to achieve the same functionality. However the mix-and-match approach can reduce user friendliness since application programming interfaces (APIs) differ across packages and dependency management is more difficult. We believe our unified package provides an attractive solution to this challenge. Table 2 documents the methods currently available in `ghostipy`.

Software Design Considerations

As previously noted, successful completion of analyses may be hampered by long computation times or lack of system memory. Specifically, algorithmic time and space complexity are major determinants for the efficiency and performance of a software method. In general it is difficult to optimize both simultaneously. For example, time complexity may be reduced by increasing hardware parallelization, at the expense of higher space complexity (memory requirements). While we sought to lower both kinds of complexity compared to existing solutions, we gave space complexity a higher priority. Stated concretely, slow computation time is primarily a nuisance, but failure to complete

	Python	Overlap save con- volution	Multi- taper method	Hilbert Trans- form	CWT	Synchro- squeezed trans- form	Out-of- core
Ghostipy	+	+	+	+	+	+	+
SciPy	+	+	-	+	-	-	-
Chronux	-	-	+	-	-	-	-
Elephant	+	+	-	+	-	-	-
Brain- storm	+	+	+	+	+	-	-
PyWt	+	-	-	-	+	+	-
Field Trip	-	+	+	+	+	-	-
MNE	+	+	+	+	+	-	-
ssqueezepy	+	-	-	-	+	+	-
MATLAB	-	+	+	+	+	+	-

Table 1. Features implemented by **ghostipy** compared to existing software.

an analysis due to insufficient memory is catastrophic.

Our design decision to prioritize space complexity was particularly critical because it directly influenced which backend library we chose for the Fast Fourier Transform (FFT), an operation used in the majority of **ghostipy**'s methods. While investigating the different options, we saw that **numpy** currently uses the **pocketfft** backend (Reinecke 2019; Van Der Walt, Colbert, and Varoquaux 2011). When accelerated with Intel's **MKL** library, it can be slightly faster than **FFTW** (Intel 2020). However, we have found **FFTW** (Frigo and Johnson 1998, 2005) superior for memory management and better suited for arbitrary length FFTs, including prime and odd numbers. An additional benefit of **FFTW** was its multithreaded capabilities (Fig. 1). We therefore selected **FFTW** as our FFT backend.

To lower space complexity we used blocked algorithms, including overlap save convolution, which is not offered in any of the standard Python numerical computing

Method	Description
<code>analytic_signal()</code>	Compute the analytic signal for a real-valued signal
<code>cwt()</code>	Compute the continuous wavelet transform
<code>estimate_taps()</code>	Estimate number of taps needed for an FIR filter
<code>filter_data_fir()</code>	Filter data using an FIR filter
<code>firdesign()</code>	Design an FIR filter
<code>get_tapers()</code>	Compute DPSS tapers
<code>group_delay()</code>	Get group delay of an FIR filter
<code>mtm_spectrogram()</code>	Use the multitaper method to generate a spectrogram
<code>mtm_spectrum()</code>	Use the multitaper method to estimate a spectrum
<code>plot_fourier_spectrogram()</code>	Plot a spectrogram generated from a Fourier-based method
<code>plot_frequency_response()</code>	Plot frequency response of a transfer function
<code>plot_wavelet_spectrogram()</code>	Plot a spectrogram generated from a wavelet-based method
<code>signal_envelope()</code>	Estimate the envelope of a real-valued signal
<code>signal_phase()</code>	Estimate the instantaneous phase of a real-valued signal
<code>wsst()</code>	Compute the wavelet synchrosqueezed transform

Table 2. Available methods in `ghostipy`.

libraries such as `numpy` or `scipy` (Van Der Walt, Colbert, and Varoquaux 2011; Virtanen et al. 2020). This approach enabled us to process very large data that could not fit in memory (also known as out-of-core processing). Throughout our code, we also employed other strategies such as in-place operations.

To lower the time complexity, we used efficient lengths of FFTs wherever possible, and we leveraged modern computing hardware by parallelizing our algorithms. For example a wavelet transform can be trivially parallelized since the transform for each scale is not dependent on other scales.

Figure 1. `ghostipy` uses `fftw` rather than `numpy` for its FFT backend. Note that when `fftw` is multithreaded, (a) the computation time can be reduced without (b) an increase in memory use.

Software and Code Accessibility

The code/software described in the paper is freely available online at

<https://github.com/kemerelab/ghostipy/>. Jupyter notebooks used to produce the

figures can be found at

<https://github.com/kemerelab/ghostipy/tree/master/examples/2021paper>. The

code and notebooks are also available as Extended Data. All results were obtained on an

Intel Core i7-4790 desktop running the Ubuntu 16.04 operating system.

FIR Filter Design

`ghostipy` provides classical signal processing capabilities such as filtering data, using the efficient overlap save convolution. Filtering data is an ubiquitous operation, but before this stage, the filter must itself be designed. While this step may appear somewhat trivial, it can make a significant difference, including the very existence of theta-gamma phase amplitude coupling (PAC) (Canolty et al. 2006; Dvorak and Fenton 2014).

Existing packages such as `scipy` and `MNE` offer a variety of FIR filter design methods (Gramfort et al. 2013; Virtanen et al. 2020). However they suffer from certain issues, as documented below:

1. Least squares method: A solution may result in a filter with magnitude response effectively zero throughout. This situation is more common when designing filters with passband relatively low compared to the sampling rate.
2. Remez exchange method: The algorithm may simply fail to converge.
3. Window method: The transition bands cannot be controlled exactly, and optimality can not be defined, as is the case for the least squares (L2 optimal) and Remez exchange (L1 optimal)

Therefore **ghostipy**'s filter design uses the method defined in (Burrus, Soewito, and Gopinath 1992) for the following reasons:

1. It is simple to design. The computational complexity is similar to that of a window method and can be implemented on embedded hardware if desired.
2. Optimality can be defined, as it is optimal in the L2 sense.
3. Transition bands can be defined exactly, and the steepness of the passband rolloff can be controlled by the spline power parameter.
4. The filter impulse response can be defined analytically. Consequently its computation does not suffer from the failure modes of the least squares or Remez exchange methods, as those must solve systems of linear equations. In other words, the design process is reliable and stable.

This method designs a low pass filter according to

$$h(n) = \frac{\sin(\omega_0 n)}{\pi n} \left[\frac{\sin(\Delta n/p)}{\Delta n/p} \right]^p \quad (1)$$

$$\omega_0 = \frac{\omega_2 + \omega_1}{2}, \quad \Delta = \omega_2 - \omega_1 \quad (2)$$

where ω_1 and ω_2 are radian frequencies defining the transition band boundaries. 98

`ghostipy` uses the low pass filter defined in 1 as a prototype to design more complicated 99
filters. As a result, users can request filters with arbitrary magnitude response. An 100
example is shown in Fig. 2. 101

```
tw = 2
p = 2
fs = 30000

numtaps = gsp.estimate_taps(fs, tw)
band_edges = [4, 6, 10, 12]
desired = [0, 1, 1, 0]
theta_filter = gsp.firdesign(numtaps, band_edges, desired,
                             fs=fs, p=p)

numtaps = gsp.estimate_taps(fs, tw)
band_edges = [10, 12, 55, 60, 100, 150, 200, 250, 300, 350]
desired = [1, 0, 0, 1, 1, 0, 0, 1, 1, 0]
arbitrary_filter = gsp.firdesign(numtaps, band_edges, desired,
                                 fs=fs, p=p)
```

Figure 2. FIR filter design. **(a)** A theta band filter designed for full bandwidth data. The specification of the transition bands allows for easy determination of critical frequencies. The -6 dB points are exactly the midpoints of the transition bands. **(b)** Filters with arbitrary pass and stop bands may also be designed.

Multitaper Method

Users often wish to perform a spectral decomposition on a signal of interest. This can be accomplished by using the multitaper method (Percival, Walden, et al. 1993; Thomson 1982). The technique is well-suited to reduce the variance of a spectrum estimate, which is particularly useful when working with noisy neural data. The spectrum estimate is obtained as an average of multiple statistically independent spectrum estimators for a discrete signal $x[n]$ with sampling frequency f_s :

$$\hat{S}_W^{mt}(k) = \frac{1}{L} \sum_{l=1}^L \hat{S}_{l,W}^{mt}(k) \quad (3)$$

$$\hat{S}_{l,W}^{mt}(k) = \frac{1}{f_s} \sum_{n=0}^{N-1} v_{l,W}[n] x[n] e^{-2\pi jkn/N} \quad (4)$$

Given the length of data N and a smoothing half-bandwidth W , the tapers $v_{l,W}[n]$ are computed by solving for vectors that satisfy the energy and orthogonality properties

$$\sum_{n=0}^{N-1} v_{l,W}[n] v_{l,W}[n] = 1 \quad (5)$$

$$\sum_{n=0}^{N-1} v_{l,W}[n] v_{m,W}[n] = 0, \quad l \neq m \quad (6)$$

For the tapers, `ghostipy` uses the discrete prolate spheroidal sequences (DPSS), which satisfy Eqns. 5 and 6 and maximize the power in the band $[-W, W]$ (Thomson 1982). An example for computing the multitapered spectrum is shown in Fig. 3.

```

psds = []
bandwidths = [10, 15, 20, 25]
for bandwidth in bandwidths:
    psd, freqs = gsp.mtm_spectrum(ripple_data,
                                  fs=1250,
                                  n_fft_threads=8,
                                  bandwidth=bandwidth)

    psds.append(psd)

```

Figure 3. Multitapered spectra. Data are from a sharp wave ripple event, where energy occurs mainly between 100 and 250 Hz. Bandwidths are (a) 10 Hz, (b) 15 Hz, (c) 20 Hz, (d) 25 Hz. Note in the code that the data sampling rate is 1250 Hz, the FFT is parallelized across 8 threads, and `ripple_data` is a 1d numpy array.

Continuous Wavelet Transform

114

Neuroscientists often use a continuous wavelet transform (CWT) to study transient

115

oscillatory activity. The CWT itself is defined in the time domain by

116

$$W(a, b) = \int_{-\infty}^{\infty} \frac{1}{\sqrt{a}} \psi^* \left(\frac{t-b}{a} \right) x(t) dt \quad (7)$$

where $\psi(\dots)$ is the mother wavelet function. The transform represents a two-dimensional
decomposition in the scale (a) and time (b) planes. In the frequency domain, the CWT is
given by the inverse Fourier transform of

$$W(a) = X(\omega)\Psi^*(a\omega) \quad (8)$$

for a given scale (a), where X and Ψ are the Fourier transforms of x and ψ , respectively.

Many mother wavelet functions have been investigated in the literature, but we have
focused on the analytic wavelets, as they are found to be superior, particularly for
estimating phase (J. Lilly and Gascard 2006; J. M. Lilly and Olhede 2008, 2012; Olhede
and Walden 2002). We have implemented the analytic Morse, Morlet, and Bump wavelets,
whose respective frequency domain definitions are

$$\Psi(a\omega) = 2 e^{-(a\omega-\omega_0)^2/2} H(\omega), \quad (9)$$

$$\Psi(a\omega) = 2 \left(\frac{e\gamma}{\beta} \right)^{\frac{\beta}{\gamma}} (a\omega)^\beta e^{-(a\omega)^\gamma} H(\omega) \quad (10)$$

$$\Psi(a\omega) = 2 e^{1-(1-(\frac{\mu-\sigma}{a})^2)^{-1}} \mathbb{1}_{(\mu-\sigma)/a, (\mu+\sigma)/a} \quad (11)$$

where $\mathbb{1}_{(\mu-\sigma)/a, (\mu+\sigma)/a}$ is the indicator function for the interval $(\mu - \sigma)/a \leq \omega \leq (\mu + \sigma)/a$
and $H(\omega)$ is the Heaviside step function. In our implementation, we use Eq. 8 to compute
the CWT.

Note that in practice the timeseries $x(t)$ is sampled, and the CWT is likewise sampled.
Then equation 8 becomes a pointwise complex multiplication of discrete Fourier transforms,

where the discretized angular frequencies ω_k are determined by

$$\omega_k = \frac{2\pi k}{N\Delta t} \quad (12)$$

where N is the number of data samples and Δt is the sampling interval.

A naive implementation of the wavelet transform (Eqn. 8) calculates untruncated wavelets the same length as the input data. This is often inefficient because it is equivalent to convolving the data with a time-domain wavelet mainly consisting of leading and trailing zeros. In our approach we exploit the fact that wavelets are finite in time and frequency, and we use an overlap-save algorithm to compute the CWT purely in the frequency domain. Note that the latter point is particularly critical: Due to the Gibbs phenomenon, using any time-domain representation of the wavelet may violate numerical analyticity for wavelet center frequencies near the Nyquist frequency. It is therefore necessary to use only the frequency domain representation of the wavelet. While we offer both traditional/naive and blockwise convolution implementations, the latter will give superior performance for longer-duration data. We believe that this is a valuable option for researchers and that this is the first tool which uses blockwise convolution to implement the CWT.

For electrophysiological data, a typical wavelet analysis will require computing Eq. 8 for 50-500 scales. This is an obvious candidate for parallelization since the wavelet transform for each scale can be computed independently of the others. We use a backend powered by **dask** to carry out the parallelization (Rocklin 2015). Users can set the number of parallel computations to execute and thereby leverage the multicore capabilities offered by modern computing hardware.

Synchrosqueezing Transform

One disadvantage of the wavelet transform is that its frequency resolution decreases as the temporal resolution increases. Strictly speaking the CWT results in information contained

in the (time, scale) plane, but a single frequency is typically assigned to each scale. 154
 Regardless, spectral smearing can be observed for at higher frequencies/lower scales. 155
 However, (Daubechies 1996; Thakur et al. 2013) showed the synchrosqueezing transform 156
 (SST) could mitigate this issue by transferring a CWT's (time, scale) plane information to 157
 the (time, frequency) plane. 158

The synchrosqueezing transform proceeds as the following. For every scale a : 159

1. Compute the CWT $W(a)$ using equation 8 160
2. Compute the partial derivative 161

$$\partial_b W(a) = j\omega X(\omega) \Psi(a\omega) \quad (13)$$

3. Compute the phase transform 162

$$\omega_f(a) = \frac{1}{2\pi} \Im \left(\frac{\partial_b W(a)}{W(a)} \right) \quad (14)$$

The phase transform contains the real frequencies each point in the CWT matrix should 163
 be assigned to. In practice the real frequency space is discretized, so the CWT points are 164
 assigned to frequency bins. Note that multiple CWT points at a given time coordinate b 165
 may map to the same frequency bin. In this situation, a given frequency bin is a simple 166
 additive accumulation of CWT points. 167

Note the similarity of the SST to the spectral reassignment algorithms in (Fitz and 168
 Fulop 2009; Gardner and Magnasco 2006). However an important distinction is that the 169
 SST only operates along the scale dimension. In addition to preserving the temporal 170
 resolution of the CWT, this makes SST data easy to work with since uniform sampling can 171
 be maintained. 172

Overall, the spectrogram methods implemented by `ghostipy` give an experimenter a 173

```
import scipy.signal as sig
fs = 1250
nperseg = 64
noverlap = 16
w = 25
f_spect, t_spect, psd_spect = sig.spectrogram(data, fs=fs,
                                                nperseg=nperseg,
                                                noverlap=noverlap)
psd_mtm, f_mtm, t_mtm = gsp.mtm_spectrogram(data, w, fs=fs,
                                              nperseg=nperseg,
                                              noverlap=noverlap)

coefs_cwt, _, f_cwt, t_cwt, _ = gsp.cwt(data, fs=fs,
                                          freq_limits=[1, 500])
coefs_wsst, _, f_wsst, t_wsst, _ = gsp.wsst(data, fs=fs,
                                              freq_limits=[1, 500],
                                              voices_per_octave=32)

psd_cwt = np.abs(coefs_cwt)**2 / fs
psd_wsst = np.abs(coefs_wsst)**2 / fs
```

Figure 4. Time-frequency decompositions. Users can leverage **(a)** scipy's spectrogram along with **(b-d)** ghostipy's methods for a richer understanding of their data. The synchrosqueezed transform in **(d)** gives the overall sharpest time and frequency resolution. Note in the code that `data` is a 1d `numpy` array, `fs` is the sampling rate, `nperseg` is the spectrogram window size in samples, `noverlap` is the number of samples overlapping in adjacent windows, and `w` is the bandwidth for the multitapered spectrogram.

more complete picture of the time-varying spectral content of neural data. Fig. 4 174
 illustrates using the `scipy` standard spectrogram method along with `ghostipy`'s methods. 175

Results 176

Example Analyses 177

An example spectrogram of local field potentials recorded in area CA1 of the rat 178
 hippocampus is depicted in Fig. 5. Clearly apparent are the theta oscillation, theta-nested 179
 gamma oscillations, and a sharp-wave-ripple, which occurs after the animal has stopped 180
 moving. 181

In addition `ghostipy` can be employed as an intermediate for a multi-step analysis. Fig. 182
 6 replicates the speed spectrogram analysis in (Kemere et al. 2013) for an animal exploring 183
 a novel and familiar environment. Fig. 7 implements the clustering of theta cycles (Zhang 184
 et al. 2019) with Morse wavelets. We have also included notebooks to replicate these 185
 example analyses. 186

Performance and Complexity 187

The calculation of the CWT is computationally intensive and consequently a good method 188
 to benchmark performance. Of the software packages listed in Table 1, only `MATLAB` offered 189
 an equivalent solution. It was thus chosen as the reference to compare our implementation 190
 against. Fig. 8 shows that our implementation results in faster computation times and 191
 better memory usage. 192

It is not entirely clear what accounts for the higher jaggedness in the `MATLAB` curves 193
 from Fig. 8. A possible explanation is that the FFT computation is less efficient for an 194
 odd-length transform, but the magnitude of the spikes in the curve is surprising given that 195
`MATLAB`'s FFT backend also uses `FFTW`. Regardless, we have demonstrated that our 196

Figure 5. CWT spectrogram of LFP. Spectrogram of local field potential recordings from area CA1 of the hippocampus of a rat during a 5-minute exploration (middle), with movement speed (top) and the raw electrophysiological signal (bottom). A number of features of the hippocampal rhythms can be noted in this example, including the pervasive theta oscillation (~ 8 Hz), theta-nested gamma oscillations (~ 60 Hz) during movement, and, towards the end, a sharp wave ripple (~ 200 Hz). Morse wavelets ($\gamma = 3$, $\beta = 10$) were used, and frequencies were limited to $[1, 250]$ Hz.

implementation is able to achieve lower time and space complexity. When using the
 functionality offered by `ghostipy`, three primary scenarios arise with regards to the sizes of
 data involved in the processing:

1. both the input and output data fit into core memory
2. the input fits into core memory but the output does not
3. neither the input nor the output fit

In all of the previous examples, we have restricted ourselves to case 1. However, with

Figure 6. Speed spectrogram. The multitaper spectrogram (5 Hz bandwidth) was computed with `ghostipy` for non-overlapping 0.5 s time bins and then z-scored for each frequency. Each time bin in the spectrogram was assigned to one of 21 logarithmically spaced speed bins spanning 0.125 to 64 cm/s. The mean PSD for each speed bin is shown for an animal exploring a (a) novel environment and a (b) familiar environment.

Figure 7. Theta cycle clustering. The Morse wavelet ($\gamma = 3, \beta = 20$) CWT was computed with `ghostipy` using 81 frequencies and subsequently divided into multiple window, where one window corresponded to one theta cycle. Each CWT sample in a window was assigned to one of 20 phase bin according to the instantaneous theta phase at that particular sample. The result was frequency-phase power profiles, which were then clustered into 4 clusters. Shown is the mean frequency-phase power profile for each cluster.

the ever-increasing sizes of data, the other cases will inevitably be encountered. Case 2 204
may arise when attempting to generate spectrograms. As the input is a single channel, 205
memory constraints are rarely an issue. For example, even a 10 hr LFP recording sampled 206

Figure 8. CWT implementation performance. Our implementation of the Morse continuous wavelet transform outperforms MATLAB’s in both (a) time and (b) space complexity. Note that MATLAB was unable to complete execution for the full range of the test parameter (data length) due to out-of-memory exceptions. The test machine was an Intel Core i7-4790 (8 hyperthreads) equipped with 32 GB RAM. In both cases, the CWT was parallelized over available CPU threads.

at 1 kHz and saved as 64-bit floating point values will require less than 300 MiB of memory. 207
 However, the size of a wavelet spectrogram computed from this data will be directly 208
 proportional to the number of scales/frequencies. For a typical range of 1 to 350 Hz at 10 209
 voices per octave, this amount to a space requirement of 85 times that of the input data. 210
 Given that this can well exceed the core memory size of a machine, **ghostipy**’s CWT 211
 routine can also accept a pre-allocated output array that is stored on disk (Fig. 9). 212

Case 3 may arise when a user wishes to filter many channels of full bandwidth data. 213
 One use case is a 1 hour recording for a 256 channel probe sampled at 30 kHz and stored as 214
 a 2-byte signed integer type; already this requires 51 GiB. Our strategy is similar to case 2, 215
 where an output array is allocated and stored on disk. As for the input, it is read in 216
 chunks, and the size of these can be chosen to lower memory usage, although potentially at 217
 a cost to computation time. The code in Fig. 10 illustrates an example: 218

Several points can be made about the scheme in Fig. 10. Our method allows for 219
 downsampling during the convolution, which can reduce the number of stages in a 220
 computational scheme. Given full bandwidth data, a traditional strategy to filter to the 221

theta band would look like the following:

1. Apply an anti-aliasing filter.
2. Downsample to obtain LFP.
3. Store the LFP to disk.
4. Apply a theta-band filter.
5. Downsample this output.
6. Save the result.

Using `ghostipy`'s method, it is not necessary to generate the intermediate LFP. To our knowledge, we do not know of other software that allows out-of-core filtering and downsampling in a single function call. The result is a simultaneous reduction in time and space complexity, by storing only the downsampled result and by filtering only once. Filtering to the theta band is now simplified to the following steps:

1. Apply a theta filter to the full bandwidth data.

```
with h5py.File(output_filepath, 'w') as outfile:

    shape, dtype = gsp.cwt(data, fs=1250, freq_limits=[1, 350],
                           describe_dims=True)

    cwt_data = outfile.create_dataset('cwt_data', shape=shape,
                                     dtype=dtype)

    gsp.cwt(data, fs=fs, freq_limits=[1, 350],
            cwt_out=cwt_data)
```

Figure 9. CWT out-of-core. Example code when the output array is too large for main memory. The CWT method is first executed as a dry run to compute the necessary array sizes. Here `data` is a 1d numpy array, and `cwt_data` is an HDF5 array created to store the results to disk.

```
import h5py
with h5py.File(input_filepath, 'r') as infile:
    with h5py.File(output_filepath, 'w') as outfile:

        ds = 300
        K = filter_delay
        N = infile['chdata'].shape[1]
        shape, dtype = gsp.filter_data_fir(infile['chdata'],
                                           theta_filter,
                                           axis=1,
                                           ds=ds,
                                           output_index_bounds=[K, K+N],
                                           describe_dims=True)

        outdata = outfile.create_dataset('theta_data',
                                         shape=shape,
                                         dtype=dtype)

        gsp.filter_data_fir(infile['chdata'],
                             theta_filter,
                             axis=1,
                             ds=ds,
                             output_index_bounds=[K, K+N],
                             outarray=outdata)
```

Figure 10. Filtering out-of-core. Filtering data from a large array stored on disk and likewise storing the output on disk. Similar to the CWT out-of-core features, the method is called once as a dry run to compute array sizes, which the user can then pass in to store the result. The filtering method also allows to correct for the delay of the filter and to downsample without storing any intermediate results. Although example uses the `h5py` library, any object that behaves like an array can be used. Here `ds` is the downsampling factor, `K` is the filter group delay, `N` is the number of samples, `infile['chdata']` is a (`n_channels`, `n_samples`) array, and `outdata` is a HDF5 array.

2. Downsample the result. 235

3. Save the result to disk. 236

Discussion 237

We have described the key features of `ghostipy` and given examples of its ease of use to 238

perform computations efficiently. Users can thus conduct exploratory spectral analyses 239

quickly across a range of parameters while reducing their concerns for running out of
memory, especially since out-of-core computation is supported for many of the methods.
Thus we believe **ghostipy** is well-suited to handle the ever-increasing size of experimental
data.

In the future we plan to improve **ghostipy** with various enhancements. For example,
currently the methods are designed to offer the user a lot of low level control over areas
such as multithreading, and to work with raw array types. However, users may desire a
higher-level API. For this reason we believe it would be a worthwhile endeavor to
incorporate our work into frameworks such as NWB (Teeters et al. 2015); this would also
facilitate more widespread adoption. There are also other analyses we could implement,
including the adaptive multitaper method (Percival, Walden, et al. 1993) and other
time-frequency reassignment techniques similar to the synchrosqueezing transform
(Daubechies, Wang, and Wu 2016).

Our primary contribution is improving the ease and speed at which data analysis can be
conducted, by developing user-friendly software implementing efficient algorithms
well-suited for large data sizes. This point is specifically demonstrated by our ability to
outperform existing solutions in space and time complexity, and to run computations even
in out-of-core memory conditions, which enables machines with 1-10s of GBs of memory to
process data on the scale of 10-100s GBs and higher. In these ways, we have increased the
accessibility of neural data analysis by enabling it to be run on hardware such as laptops, a
scenario that often was not previously possible.

Finally, the software we developed has a much larger potential impact than the scope
described in this paper. Although many of the examples given in this paper were specific
to extracellular rodent hippocampal data, the functionality we implemented is intentionally
generic and applicable to many fields. As an example, our code can easily be adapted for
use in real-time processing, whether running on embedded hardware or on a laptop in a

clinical EEG setting. Given the functionality already developed and the full scope of our
work, we are optimistic that **ghostipy** can help accelerate modern scientific progress.

References

- Bokil, H., P. Andrews, J. E. Kulkarni, S. Mehta, and P. P. Mitra (2010). Chronux: a platform for analyzing neural signals. In: *Journal of Neuroscience Methods* 192.1, pp. 146–151.
- Burrus, C. S., A. W. Soewito, and R. A. Gopinath (1992). Least squared error FIR filter design with transition bands. In: *IEEE Transactions on Signal Processing* 40.6, pp. 1327–1340.
- Canolty, R. T., E. Edwards, S. S. Dalal, M. Soltani, S. S. Nagarajan, H. E. Kirsch, M. S. Berger, N. M. Barbaro, and R. T. Knight (2006). High gamma power is phase-locked to theta oscillations in human neocortex. In: *Science* 313.5793, pp. 1626–1628.
- Daubechies, I. (1996). A nonlinear squeezing of the continuous wavelet transform based on auditory nerve models. In: *Wavelets in Medicine and Biology*, pp. 527–546.
- Daubechies, I., Y. Wang, and H.-t. Wu (2016). ConceFT: Concentration of frequency and time via a multitapered synchrosqueezed transform. In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 374.2065, p. 20150193.
- Dvorak, D. and A. A. Fenton (2014). Toward a proper estimation of phase–amplitude coupling in neural oscillations. In: *Journal of Neuroscience Methods* 225, pp. 42–56.
- Fitz, K. R. and S. A. Fulop (2009). A unified theory of time-frequency reassignment. In: *arXiv preprint arXiv:0903.3080*.

- Frigo, M. and S. G. Johnson (1998). FFTW: An adaptive software architecture for the
FFT. In: *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech
and Signal Processing, ICASSP'98 (Cat. No. 98CH36181)*. Vol. 3. IEEE, pp. 1381–1384.
- (2005). The design and implementation of FFTW3. In: *Proceedings of the IEEE* 93.2,
pp. 216–231.
- Gardner, T. J. and M. O. Magnasco (2006). Sparse time-frequency representations. In:
Proceedings of the National Academy of Sciences 103.16, pp. 6094–6099.
- Gramfort, A., M. Luessi, E. Larson, D. A. Engemann, D. Strohmeier, C. Brodbeck, R. Goj,
M. Jas, T. Brooks, L. Parkkonen, et al. (2013). MEG and EEG data analysis with
MNE-Python. In: *Frontiers in Neuroscience* 7, p. 267.
- Intel (2020). *Performance Benchmarks*.
<https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library/benchmarks/>
[Online; accessed 2020-06-09].
- Kemere, C., M. F. Carr, M. P. Karlsson, and L. M. Frank (2013). Rapid and Continuous
Modulation of Hippocampal Network State during Exploration of New Places. In: *PLOS*
ONE 8.9, pp. 1–16.
- Lee, G., R. Gommers, F. Waselewski, K. Wohlfahrt, and A. O’Leary (2019). PyWavelets: A
Python package for wavelet analysis. In: *Journal of Open Source Software* 4.36, p. 1237.
- Lilly, J. and J.-C. Gascard (2006). Wavelet ridge diagnosis of time-varying elliptical signals
with application to an oceanic eddy. In: *Nonlinear Processes in Geophysics* 13.5,
pp. 467–483.
- Lilly, J. M. and S. C. Olhede (2008). Higher-order properties of analytic wavelets. In: *IEEE*
Transactions on Signal Processing 57.1, pp. 146–160.
- (2012). Generalized Morse wavelets as a superfamily of analytic wavelets. In: *IEEE*
Transactions on Signal Processing 60.11, pp. 6036–6041.

- Olhede, S. C. and A. T. Walden (2002). Generalized morse wavelets. In: *IEEE Transactions on Signal Processing* 50.11, pp. 2661–2670.
- Oostenveld, R., P. Fries, E. Maris, and J.-M. Schoffelen (2011). FieldTrip: open source software for advanced analysis of MEG, EEG, and invasive electrophysiological data. In: *Computational Intelligence and Neuroscience* 2011.
- OverLordGoldDragon (2020). ssqueezepy. In: *GitHub. Note:* <https://github.com/OverLordGoldDragon/ssqueezepy/>. DOI: [10.5281/zenodo.5080514](https://doi.org/10.5281/zenodo.5080514).
- Percival, D. B., A. T. Walden, et al. (1993). *Spectral analysis for physical applications*. cambridge university press.
- Reinecke, M. (2019). *Performance Benchmarks*. <https://gitlab.mpcdf.mpg.de/mtr/pocketfft>. [Online; accessed 2020-06-11].
- Rocklin, M. (2015). Dask: Parallel computation with blocked algorithms and task scheduling. In: *Proceedings of the 14th Python in Science Conference*. Vol. 126. Citeseer.
- Tadel, F., E. Bock, G. Niso, J. C. Mosher, M. Cousineau, D. Pantazis, R. M. Leahy, and S. Baillet (2019). MEG/EEG group analysis with brainstorm. In: *Frontiers in Neuroscience* 13, p. 76.
- Teeters, J. L., K. Godfrey, R. Young, C. Dang, C. Friedsam, B. Wark, H. Asari, S. Peron, N. Li, A. Peyrache, et al. (2015). Neurodata without borders: creating a common data format for neurophysiology. In: *Neuron* 88.4, pp. 629–634.
- Thakur, G., E. Brevdo, N. S. Fućkar, and H.-T. Wu (2013). The synchrosqueezing algorithm for time-varying spectral analysis: Robustness properties and new paleoclimate applications. In: *Signal Processing* 93.5, pp. 1079–1094.
- Thomson, D. J. (1982). Spectrum estimation and harmonic analysis. In: *Proceedings of the IEEE* 70.9, pp. 1055–1096.

- Van Der Walt, S., S. C. Colbert, and G. Varoquaux (2011). The NumPy array: a structure 338
for efficient numerical computation. In: *Computing in Science & Engineering* 13.2, 339
pp. 22–30. 340
- Virtanen, P., R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, 341
E. Burovski, P. Peterson, W. Weckesser, J. Bright, et al. (2020). SciPy 1.0: fundamental 342
algorithms for scientific computing in Python. In: *Nature Methods* 17.3, pp. 261–272. 343
- Yegenoglu, A., D. Holstein, L. D. Phan, M. Denker, A. Davison, and S. Grün (2015). 344
Elephant—open-source tool for the analysis of electrophysiological data sets. Tech. rep. 345
Computational and Systems Neuroscience. 346
- Zhang, L., J. Lee, C. Rozell, and A. C. Singer (2019). Sub-second dynamics of 347
theta-gamma coupling in hippocampal CA1. In: *eLife* 8, e44320. 348