Novel Tools and Methods

GhostiPy: An Efficient Signal Processing and Spectral Analysis Toolbox for Large Data

Joshua P. Chu, and Caleb T. Kemere

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

Department of Electrical and Computer Engineering, Rice University, Houston, Texas 77251-1892

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 **Py**thon), 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.

Key words: local field potential; oscillations; signal processing; spectral analysis

Significance Statement

Because of 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 (e.g., 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 computer servers. Overall, GhostiPy improves experimental throughput by increasing the portability of analyses.

Received May 5, 2021; accepted August 31, 2021; First published September 23, 2021.

The authors declare no competing financial interests.

Author contributions: J.P.C. and C.T.K. designed research; J.P.C. performed research; J.P.C. and C.T.K. analyzed data; J.P.C. and C.T.K. wrote the paper.

The development of GhostiPy was supported by the National Science Foundation (Grant NSF CBET1351692) and the National Institute of Neurological Diseases and Strokes (Grant R01-NS-115233).

Acknowledgements: We thank Shayok Dutta (Rice University), Andres Grosmark and Gyorgi Buzsaki (NYU, New York, NY), Loren Frank and Mattias Karlsson (UCSF), and the CRCNS.org data archive for sharing data used in example analyses.

Correspondence should be addressed to Caleb T. Kemere at caleb.kemere@rice.edu.

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

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 that may manifest as the following scenarios: (1) analysis code takes a long time to complete (high time complexity); and (2) code is unable to complete because of insufficient memory on the hardware (high spatial complexity). Moreover, the scientist may have difficulty finding existing tools that address both 1 and 2 and 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

Table 1: Features implemented by GhostiPy compared with existing software

	Python	Overlap save convolution	Multitaper method	Hilbert transform	CWT	Synchrosqueezed transform	Out-of-core
Ghostipy	+	+	+	+	+	+	+
SciPy	+	+	–	+	–	–	–
Chronux	–	–	+	–	–	–	–
Elephant	+	+	–	+	–	–	–
BrainStorm	+	+	+	+	+	–	–
PyWt	+	–	–	–	+	+	–
Field Trip	–	+	+	+	+	–	–
MNE	+	+	+	+	+	–	–
ssqueezepy	+	–	–	–	+	+	–
MATLAB	–	+	+	+	+	+	–

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 article, we first describe the software design principles of GhostiPy to increase efficiency. We then elaborate on featured methods along with code samples illustrating the 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 (OverLordGold Dragon, https://github.com/OverLordGoldDragon/ssqueezepy/; Bokil et al., 2010; Oostenveld et al., 2011; Gramfort et al., 2013; Yegenoglu et al., 2015; Lee et al., 2019; Tadel et al., 2019; Virtanen et al., 2020) 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 is a major determinant 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 with existing solutions, we gave space complexity a higher priority. Stated concretely, slow computation time is primarily a nuisance, but failure to complete an analysis because of 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 the GhostiPy methods. While investigating the different options, we saw that numpy currently uses the pocketfft backend (https://gitlab.mpcdf.mpg.de/mtr/pocketfft; Van Der Walt et al., 2011). When accelerated with the Intel MKL library, it can be slightly faster than FFTW (https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library/benchmarks.html). However, we have found FFTW (Frigo

Table 2: Available methods in GhostiPy

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

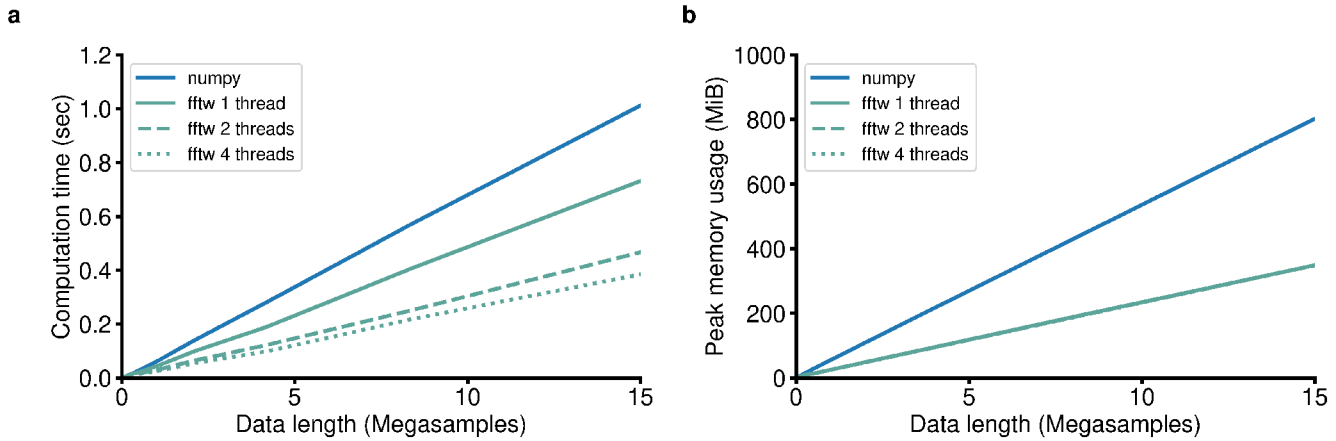

Figure 1. GhostiPy uses fftw rather than numpy for its FFT backend. ***a***, ***b***, Note that when fftw is multithreaded, the computation time can be reduced (***a***) without an increase in memory use (***b***).

```
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)
```

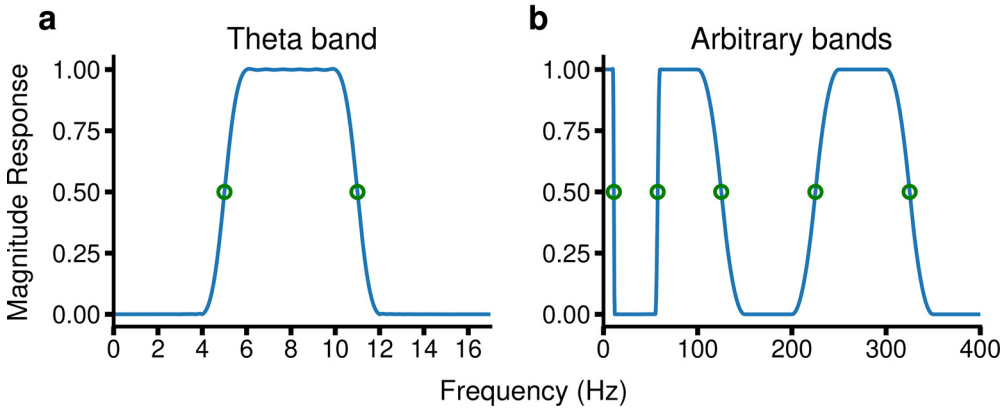

Figure 2. FIR filter design. ***a***, A theta-band filter designed for full bandwidth data. The specification of the transition bands allows for easy determination of critical frequencies. The −6 dB points are exactly the midpoints of the transition bands. ***b***, Filters with arbitrary pass and stop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

and Johnson, 1998, 2005) to be superior for memory management and better suited for FFTs of arbitrary length, 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 libraries such as numpy or scipy (Van Der Walt et al., 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 used 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.

```
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)
```

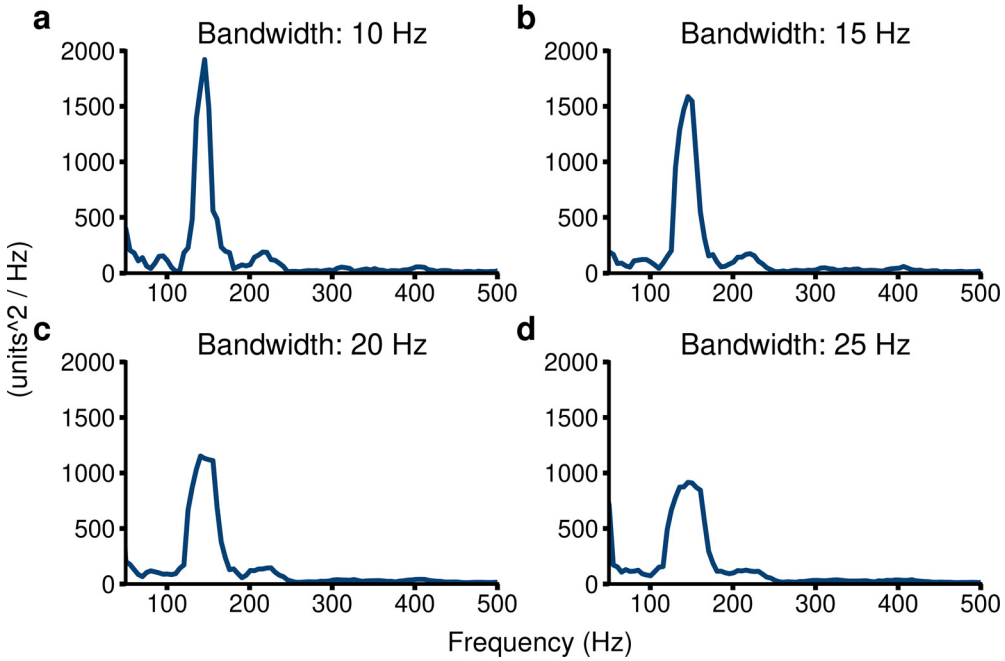

Figure 3. Multitapered spectra. Data are from a sharp wave ripple event, where energy occurs mainly between 100 and 250 Hz. ***a–d***, Bandwidths are 10 Hz (***a***), 15 Hz (***b***), 20 Hz (***c***), and 25 Hz (***d***). Note in the code that the data-sampling rate is 1250 Hz, the FFT is parallelized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

Finite impulse response filter design

GhostiPy provides classical signal processing capabilities such as filtering data, using the efficient overlap save convolution. Filtering data is a 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 (Canolty et al., 2006; Dvorak and Fenton, 2014).

Existing packages such as scipy and MNE offer a variety of finite impulse response (FIR) filter design methods (Gramfort et al., 2013; Virtanen et al., 2020). However, some methods suffer from the following issues. (1) Using the least-squares method, a solution may result in a filter with a magnitude response effectively of zero throughout. This situation is more common when designing filters with passband relatively low compared with the sampling rate. (2) Using the Remez exchange method, the algorithm may simply fail to converge. (3) Using the window method, the transition bands cannot be controlled exactly, and optimality cannot be defined, as is the case for the least-squares (L2 optimal) and Remez exchange (L1 optimal) methods.

Therefore, the GhostiPy filter design uses the method defined in the study by Burrus et al. (1992) for the following reasons: (1) it is simple to design, and 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; and (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 the following:

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

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

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

GhostiPy uses the low-pass filter defined in Equation 1 as a prototype to design more complicated filters. As

```
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
```

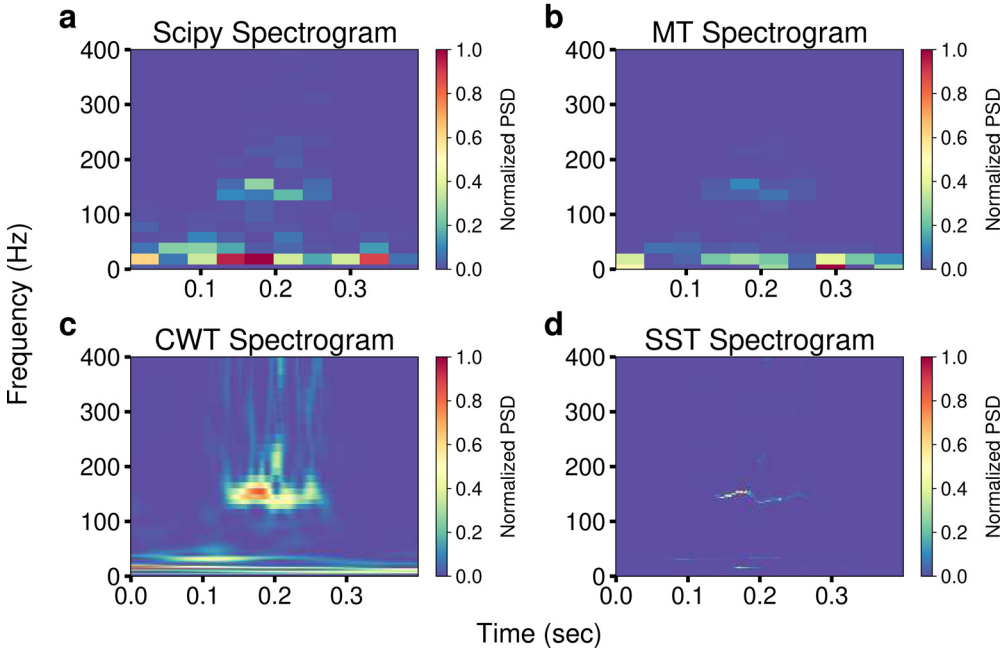

Figure 4. Time–frequency decompositions. ***a–c***, Users can leverage the scipy spectrogram (***a***) along with the methods of GhostiPy (***b–d***) for a richer understanding of their data. The synchrosqueezed transform in ***d*** gives the overall sharpest time and frequency resolution. Note in the code that data are 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.

a result, users can request filters with arbitrary magnitude response. An example is shown in Figure 2.

Multitaper method

Users often wish to perform a spectral decomposition on a signal of interest. This can be accomplished by using the multitaper method (Thomson, 1982; Percival and Walden, 1993). 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, as follows:

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

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

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, as follows:

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

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

For the tapers, GhostiPy uses the discrete prolate spheroidal sequences (DPSSs), which satisfy Equations 5 and 6 and maximize the power in the band [– W,W]

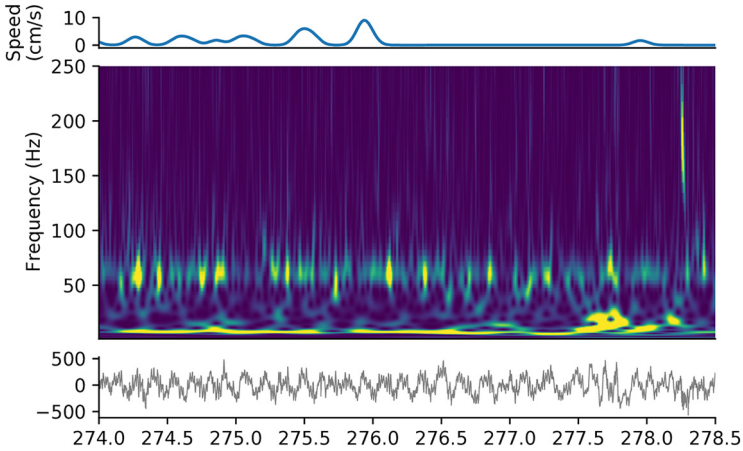

Figure 5. CWT spectrogram of LFP. Spectrogram of local field potential recordings from area CA1 of the hippocampus of a rat during a 5 min exploration (middle), with movement speed (top) and the raw electrophysiological signal (bottom). A number of features of the hippocampal rhythms can be noted in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

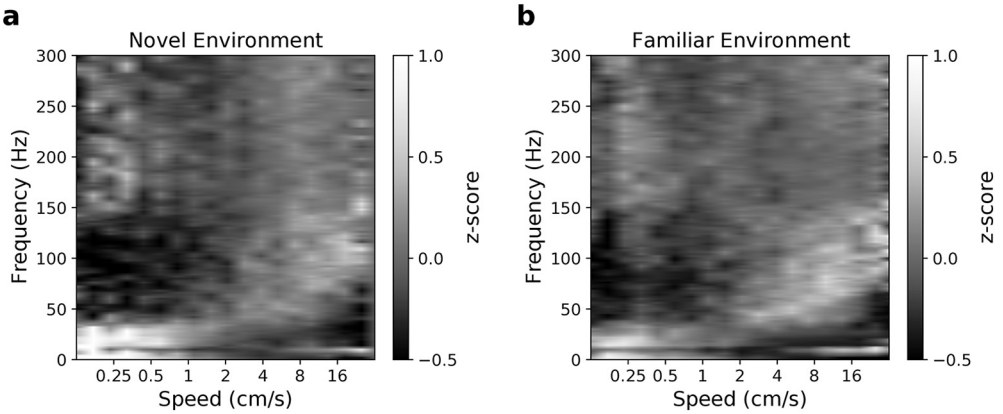

Figure 6. Speed spectrogram. The multitaper spectrogram (bandwidth, 5 Hz) was computed with GhostiPy for nonoverlapping 0.5 s time bins and then *z* scored for each frequency. Each time bin in the spectrogram was assigned to 1 of 21 logarithmically spaced speed bins spanning 0.125–64 cm/s. ***a***, ***b***, The mean PSD for each speed bin is shown for an animal exploring a novel environment (***a***) and a familiar environment (***b***).

(Thomson, 1982). An example for computing the multitapered spectrum is shown in Figure 3.

Continuous wavelet transform

Neuroscientists often use a continuous wavelet transform (CWT) to study transient oscillatory activity. The CWT itself is defined in the time domain by the following:

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

where $\psi(...)$ 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 the following:

$$W(a) = X(\omega)\Psi^*(a\omega), \tag{8}$$

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 the phase (Olhede and Walden, 2002; Lilly and Gascard, 2006; Lilly and Olhede, 2009, 2012. We have implemented the analytic Morse, Morlet, and Bump wavelets, whose respective frequency domain definitions are as follows:

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

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

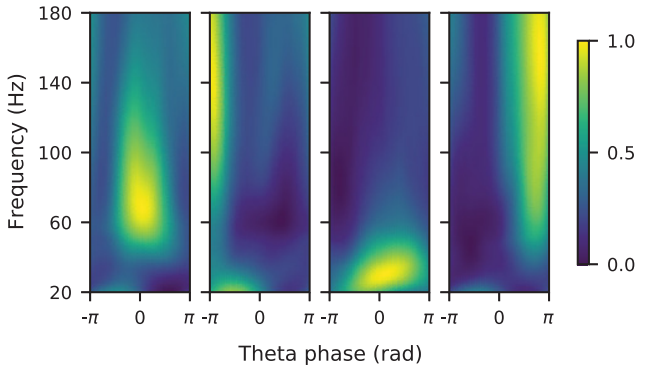

Figure 7. Theta cycle clustering. The Morse wavelet ($\gamma = 3$, $\beta = 20$) CWT was computed with GhostiPy using 81 frequencies and was subsequently divided into multiple windows, where one window corresponded to one theta cycle. Each CWT sample in a window was assigned to 1 of 20 phase bins according to the instantaneous theta phase at that particular sample. The result was frequency–phase power profiles, which were then clustered into four clusters. Shown is the mean frequency–phase power profile for each cluster. As in Zhang et al., 2019, these use the hc-11 dataset from CRCNS.org (Grossmark and Buzaki, 2016, Grossmark et al., 2016).

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

where $1_{(\mu-\sigma)/a,(\mu+\sigma)/a}$ 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 Equation 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 the following:

$$\omega_k = \frac{2\pi k}{N\Delta t}, \tag{12}$$

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

A naive implementation of the wavelet transform (Eq. 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: because of 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 that uses blockwise convolution to implement the CWT.

For electrophysiological data, a typical wavelet analysis will require computing Equation 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. Regardless, spectral smearing can be observed at higher frequencies/lower scales. However Daubechies (1996) and Thakur et al. (2013) showed the synchrosqueezing transform (SST) could mitigate this issue by transferring a CWT (time, scale) plane information to the (time, frequency) plane.

The synchrosqueezing transform proceeds as follows. For every scale a: compute the CWT $W(a)$ using Equation 8, compute the following partial derivative:

$$\partial_b W(a) = j\omega X(\omega)\Psi(a\omega), \tag{13}$$

and compute the following phase transform:

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

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

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

Overall, the spectrogram methods implemented by GhostiPy give an experimenter a more complete picture of the time-varying spectral content of neural data. Figure 4 illustrates this using the scipy standard spectrogram method along with the GhostiPy methods.

Data availability

The code/software described in the article is freely available online at https://github.com/kemerelab/ghostipy/.

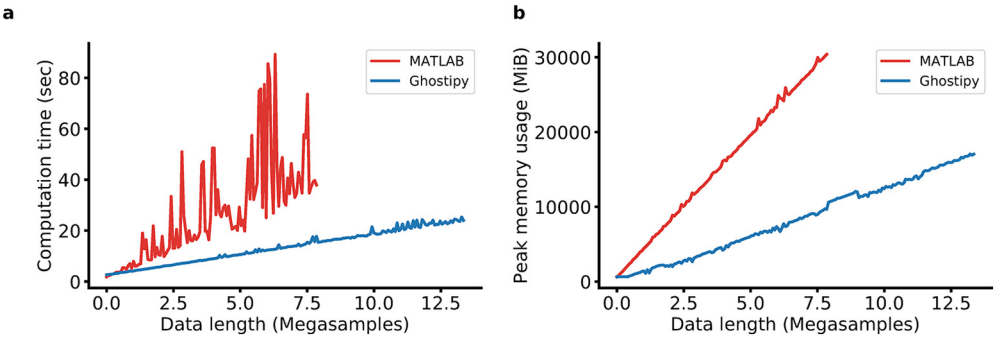

Figure 8. CWT implementation performance. ***a***, ***b***, Our implementation of the Morse continuous wavelet transform outperforms MATLAB in both time (***a***) and space complexity (***b***). Note that MATLAB was unable to complete execution for the full range of the test parameter (data length) because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

Jupyter Notebook, which can be found at https://github.com/kemerelab/ghostipy/tree/master/examples/2021paper, was used to produce the figures. The code and notebooks are also available as Extended Data 1 and Extended Data 2. All results were obtained on an Intel Core i7-4790 desktop computer running the Ubuntu 16.04 operating system.

Results

Example analyses

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

In addition, GhostiPy can be used as an intermediate for a multistep analysis. Figure 6 replicates the speed spectrogram analysis in the study by Kemere et al. (2013) for an animal exploring a novel and a familiar environment (Mattias et al., 2015). Figure 7 implements the clustering of theta cycles (Zhang et al., 2019) with Morse wavelets. We have also included notebooks to replicate these example analyses.

Performance and complexity

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

It is not entirely clear what accounts for the higher jaggedness in the MATLAB curves from Figure 8. A possible explanation is that the FFT computation is less efficient for an odd-length transform, but the magnitude of the spikes in the curve is surprising given that the FFT backend of MATLAB also uses FFTW. Regardless, we have demonstrated that our implementation is able to achieve lower time and space complexity. When using the functionality offered by GhostiPy, the following three primary scenarios arise with regard 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; and (3) neither the input nor the output fit.

In all of the previous examples, we have restricted ourselves to case 1. However, with the ever-increasing sizes of data, the other cases will inevitably be encountered. Case 2 may arise when attempting to generate spectrograms. As the input is a single channel, memory constraints are rarely an issue. For example, even a 10 h local field potential (LFP) recording sampled at 1 kHz and saved as 64 bit floating point values will require $<$300 mebibytes (MiB) of memory. However, the size of a wavelet spectrogram computed from these data will be directly proportional to the number of scales/frequencies. For a typical range of 1–350 Hz at 10 voices per octave, this amounts to a space requirement of 85 times that of the input data. Given that this can well exceed the core memory size of a machine, the GhostiPy CWT routine can also accept a preallocated output array that is stored on disk (Fig. 9).

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

Several points can be made about the scheme in Figure 10. Our method allows for downsampling during the convolution, which can reduce the number of stages in a computational scheme. Given full bandwidth data, a traditional strategy to filter to the theta band would look like the following: (1) apply an antialiasing filter; (2) downsample to obtain LFP; (3) store the LFP to disk; (4) apply a theta-band filter; (5) downsample this output; and (5) save the result.

Using the GhostiPy 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

```
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 are a 1D numpy array, and cwt_data are 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 the 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 an HDF5 array.

steps: (1) apply a theta filter to the full bandwidth data; (2) downsample the result; and (3) save the result to disk.

Discussion

We have described the key features of GhostiPy and given examples of its ease of use to perform computations efficiently. Users can thus conduct exploratory spectral analyses 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 and Walden, 1993) and other time–frequency reassignment techniques similar to the synchrosqueezing transform (Daubechies et al., 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 1–10 s of of GBs of memory to process data on the scale of 10–100 s of 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 a laptop computer, 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 article. Although many of the examples given in this article 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 computer 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, Andrews P, Kulkarni JE, Mehta S, Mitra PP (2010) Chronux: a platform for analyzing neural signals. J Neurosci Methods 192:146–151.

Burrus CS, Soewito AW, Gopinath RA (1992) Least squared error FIR filter design with transition bands. IEEE Trans Signal Process 40:1327–1340.

Canolty RT, Edwards E, Dalal SS, Soltani M, Nagarajan SS, Kirsch HE, Berger MS, Barbaro NM, Knight RT (2006) High gamma power is phase-locked to theta oscillations in human neocortex. Science 313:1626–1628.

Daubechies I (1996) A nonlinear squeezing of the continuous wavelet transform based on auditory nerve models. In: Wavelets in medicine and biology (Aldroubi A, Unser MA, eds), pp 527–546. Boca Raton, FL: CRC.

Daubechies I, Wang Y, Wu H-t (2016) ConceFT: concentration of frequency and time via a multitapered synchrosqueezed transform. Phil Trans R Soc A 374:20150193.

Dvorak D, Fenton AA (2014) Toward a proper estimation of phase–amplitude coupling in neural oscillations. J Neurosci Methods 225:42–56.

Fitz KR, Fulop SA (2009) A unified theory of time-frequency reassignment. arXiv:0903.3080.

Frigo M, Johnson SG (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: May 12-15, 1998, Washington state convention and Trade Center, Seattle, WA (USA), Vol 3, pp 1381–1384. Piscataway, NJ: IEEE.

Frigo M, Johnson SG (2005) The design and implementation of FFTW3. Proc IEEE 93:216–231.

Gardner TJ, Magnasco MO (2006) Sparse time-frequency representations. Proc Natl Acad Sci U S A 103:6094–6099.

Gramfort A, Luessi M, Larson E, Engemann DA, Strohmeier D, Brodbeck C, Goj R, Jas M, Brooks T, Parkkonen L, Hämäläinen M (2013) MEG and EEG data analysis with MNE-Python. Front Neurosci 7:267.

Grosmark AD, Buzsáki G (2016) Diversity in neural firing dynamics supports both rigid and learned hippocampal sequences. Science 351:1440–1443.

Grosmark AD, Long J, Buzsáki G (2016) Recordings from hippocampal area CA1, PRE, during and POST novel spatial learning. CRCNS.org. http://dx.doi.org/10.6080/K0862DC5.

Kemere C, Carr MF, Karlsson MP, Frank LM (2013) Rapid and continuous modulation of hippocampal network state during exploration of new places. PLoS One 8:e73114.

Lee G, Gommers R, Waselewski F, Wohlfahrt K, O'Leary A (2019) PyWavelets: a Python package for wavelet analysis. J Open Source Softw 4:1237.

Lilly JM, Gascard J-C (2006) Wavelet ridge diagnosis of time-varying elliptical signals with application to an oceanic eddy. Nonlin Processes Geophys 13:467–483.

Lilly JM, Olhede SC (2009) Higher-order properties of analytic wavelets. IEEE Trans Signal Process 57:146–160.

Lilly JM, Olhede SC (2012) Generalized Morse wavelets as a superfamily of analytic wavelets. IEEE Trans Signal Process 60:6036–6041.

Mattias K, Margaret C, Frank LM (2015) Simultaneous extracellular recordings from hippocampal areas CA1 and CA3 (or MEC and CA1) from rats performing an alternation task in two W-shapped tracks that are geometrically identically but visually distinct. CRCNS.org. Available at http://dx.doi.org/10.6080/K0NK3BZJ.

Olhede SC, Walden AT (2002) Generalized morse wavelets. IEEE Trans Signal Process 50:2661–2670.

Oostenveld R, Fries P, Maris E, Schoffelen J-M (2011) FieldTrip: open source software for advanced analysis of MEG, EEG, and invasive electrophysiological data. Comput Intell Neurosci 2011:156869.

Percival DB, Walden AT (1993) Spectral analysis for physical applications. Cambridge, UK: Cambridge UP.

Rocklin M (2015) Dask: parallel computation with blocked algorithms and task scheduling. In: Proceedings of the 14th Python in science conference (SCIPY 2015), Vol 126 (Huff K, Bergstra J, eds). Austin, TX: SciPy Developers. [10.25080/Majora-7b98e3ed-013]

Tadel F, Bock E, Niso G, Mosher JC, Cousineau M, Pantazis D, Leahy RM, Baillet S (2019) MEG/EEG group analysis with brainstorm. Front Neurosci 13:76.

Teeters JL, Godfrey K, Young R, Dang C, Friedsam C, Wark B, Asari H, Peron S, Li N, Peyrache A, Denisov G, Siegle JH, Olsen SR, Martin C, Chun M, Tripathy S, Blanche TJ, Harris K, Buzsáki G, Koch C, et al. (2015) Neurodata without borders: creating a common data format for neurophysiology. Neuron 88:629–634.

Thakur G, Brevdo E, Fučkar NS, Wu H-T (2013) The synchrosqueezing algorithm for time-varying spectral analysis: robustness properties and new paleoclimate applications. Signal Processing 93:1079–1094.

Thomson DJ (1982) Spectrum estimation and harmonic analysis. Proc IEEE 70:1055–1096.

Van Der Walt S, Colbert SC, Varoquaux G (2011) The NumPy array: a structure for efficient numerical computation. Comput Sci Eng 13:22–30.

Virtanen P, Gommers R, Oliphant TE, Haberland M, Reddy T, Cournapeau D, Burovski E, Peterson P, Weckesser W, Bright J, van der Walt SJ, Brett M, Wilson J, Millman KJ, Mayorov N, Nelson ARJ, Jones E, Kern R, Larson E, Carey CJ, et al. (2020) SciPy 1.0: fundamental algorithms for scientific computing in Python. Nat Methods 17:261–272.

Yegenoglu A, Holstein D, Phan LD, Denker M, Davison A, Grun S (2015) Elephant–open-source tool for the analysis of electrophysiological data sets. Tech Rep Computational and Systems Neuroscience. Available at https://juser.fz-juelich.de/record/255984.

Zhang L, Lee J, Rozell C, Singer AC (2019) Sub-second dynamics of theta-gamma coupling in hippocampal CA1. eLife 8:e44320.