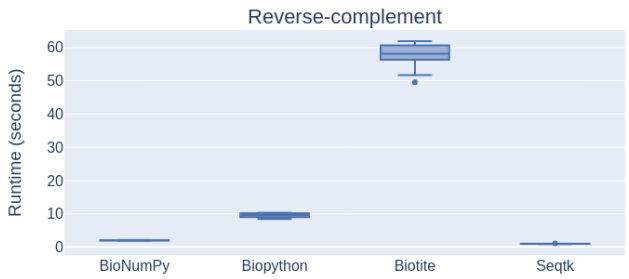
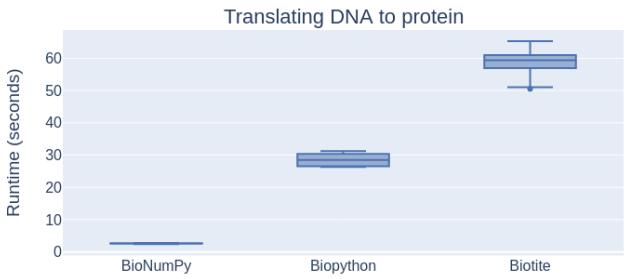
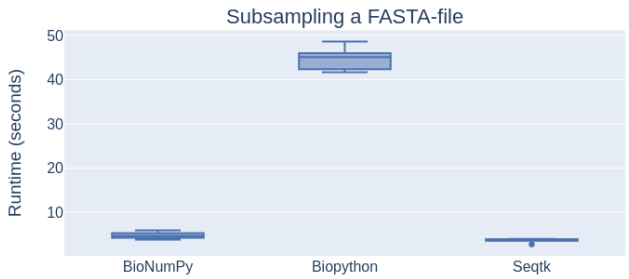
Supplementary material for: Array programming for Biology

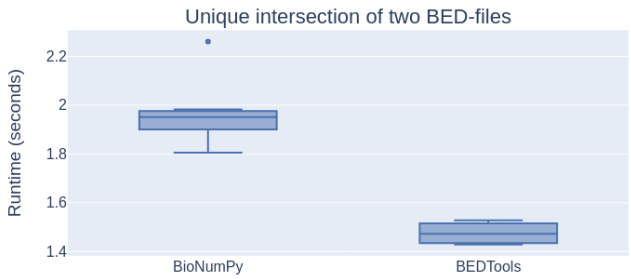
Benchmarks

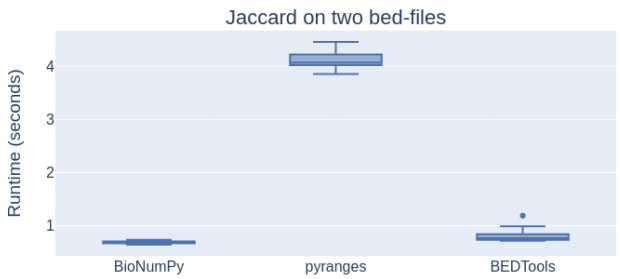
We compare the speed of BioNumPy against other existing Python packages and commonly used non-Python tools on a set of typical bioinformatics tasks. As seen in Figure 1, we find that BioNumPy is generally considerably faster than vanilla Python solutions, as well as the commonly used Python packages BioPython and Biotite, which mostly rely on Python for-loops to perform operations on datasets. On problems where designated efficient bioinformatics tools are widely used (intersection of BED-files, kmer counting and VCF operations), we find that BioNumPy is close to, or as efficient as, tools written in C/C++ (BEDTools [1], Jellyfish [2] and BCFTools [3]). A Snakemake pipeline for reproducing the results can be found at https://github.com/bionumpy/bionumpy/tree/main/benchmarks, along with an open invitation to expand the benchmark with additional tools and cases.

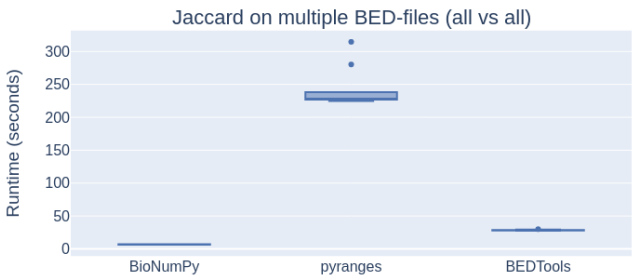

Reverse-complement
Runtime (seconds)
0
10
20
30
40
50
60
BioNumPy
Biopython
Biotite
Seqtk

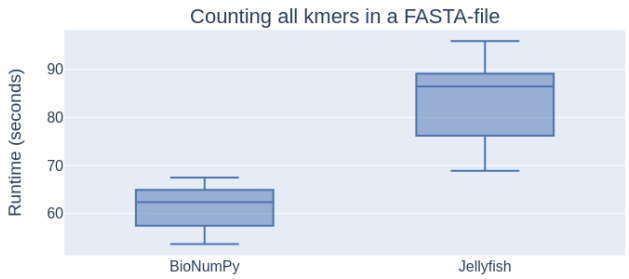

Translating DNA to protein
Runtime (seconds)
0
10
20
30
40
50
60
BioNumPy
Biopython
Biotite

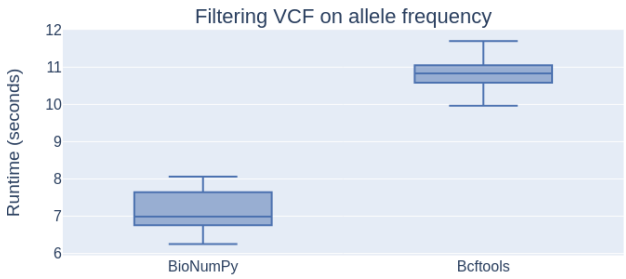

Subsampling a FASTA-file
Runtime (seconds)
10
20
30
40
50
BioNumPy
Biopython
Seqtk

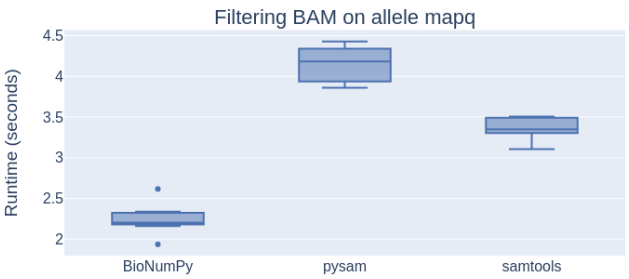

Unique intersection of two BED-files
Runtime (seconds)
1.4
1.6
1.8
2
2.2
BioNumPy
BEDTools


Jaccard on two bed-files
Runtime (seconds)
1
2
3
4
BioNumPy
pyranges
BEDTools


Jaccard on multiple BED-files (all vs all)
Runtime (seconds)
0
50
100
150
200
250
300
BioNumPy
pyranges
BEDTools


Counting all kmers in a FASTA-file
Runtime (seconds)
60
70
80
90
BioNumPy
Jellyfish


Filtering VCF on allele frequency
Runtime (seconds)
6
7
8
9
10
11
12
BioNumPy
Bcftools


Filtering BAM on allele mapq
Runtime (seconds)
2
2.5
3
3.5
4
4.5
BioNumPy
pysam
samtools

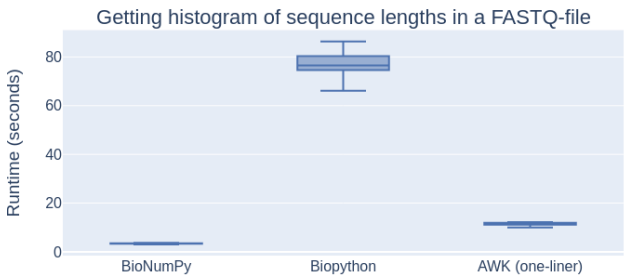

Getting histogram of sequence lengths in a FASTQ-file
Runtime (seconds)
0
20
40
60
80
BioNumPy
Biopython
AWK (one-liner)

Figure 1: Benchmarking BioNumPy against other tools and methods on various typical bioinformatics tasks.

Reproducing a machine learning benchmark using BioNumPy

To show how BioNumPy can be used to easily process and parse various biology data formats, we used BioNumPy to reproduce a recently published benchmark of a machine learning method [4]. In the original work, the authors are using a combination of custom Python code and common bioinformatics tools (such as BCFTools [3]) to extract sequences around the transcription start sites of genes, which are used as input to a machine learning method. This preprocessing step takes as input various file formats (GTF, FASTA, VCF). Due to the external dependencies and combination of scripts that need to be manually run and tied together, the current code is not runnable and the results are not directly reproducible.

We have forked the original repository and replaced all this code with a single, readable Python script that uses BioNumPy to preprocess and combine the various data files. We believe this shows that BioNumPy can be used to easily and cleanly integrate various biological datasets where it before was common to use a combination of scripts and tools, which easily leads to non-reproducible code.

In addition to being reproducible, the BioNumPy code enables several advantages. The code is reusable for other settings. It is straightforward to run the Python script with another set of genes, or another window size. It is relatively easy to adapt the script to be more robust. Some transcription start site windows might fall outside of the chromosome boundaries or include 'N's, both of which would break the current code. However, it is straightforward to handle this in the single Python script. Lastly, it is possible to try different variations of the script to see if it improves the performance. For instance, the current scripts return all the sequences on the forward strand, but it might be better to return the reverse complement for genes that are on the negative strand. Lastly, one can use the full range of modularization in Python. From functions and classes to modules and published packages installable by PiP. This makes it easier to reuse the code in other similar projects. In conclusion, BioNumPy enables efficient scripts for handling genomic data that are reproducible, reusable, adaptable, and modularizable.

Our fork is available at https://github.com/knutdrand/enformer_assesment_reproduction.

BioNumPy Implementation details

BioNumPy internally stores sequence data (e.g. nucleotides or amino acids) as numeric values, allowing the use of standard NumPy arrays for data representation and processing. A key way BioNumpy achieves high performance is by storing multiple data entries in shared NumPy arrays. To illustrate the benefit of this approach, consider the example where we want to count the number of Gs and Cs in a large set of DNA sequences. Existing Python packages like BioPython and Biotite, do this by iterating over the sequences using Python for-loops, which is slow when the number of sequences is large. BioNumPy, however, stores all sequences in only one or a few shared NumPy arrays (Figure 3a), meaning that vectorized NumPy operations can be used to do the counting in a fraction of the time.

Storing multiple elements in shared arrays is trivial if the elements all have the same size, since a matrix representation can be used. However, for biological data, it is common that data elements vary in size. For instance, sequences in FASTA files are rarely all of the exact same size. BioNumPy uses the RaggedArray data structure from the npstructures package (https://github.com/bionumpy/npstructures, developed in tandem with BioNumPy) to tackle this problem (Figure 2). The RaggedArray can be seen as a matrix where rows can have different lengths.

The npstructures RaggedArray implementation is compatible with most common NumPy operations, like indexing (Figure 2 b), vectorized operations (Figure 2 c), and reductions (Figure 2 d). As far as possible, objects in BioNumPy follow the array interoperability protocols defined by NumPy (https://numpy.org/doc/stable/user/basics.interoperability.html)

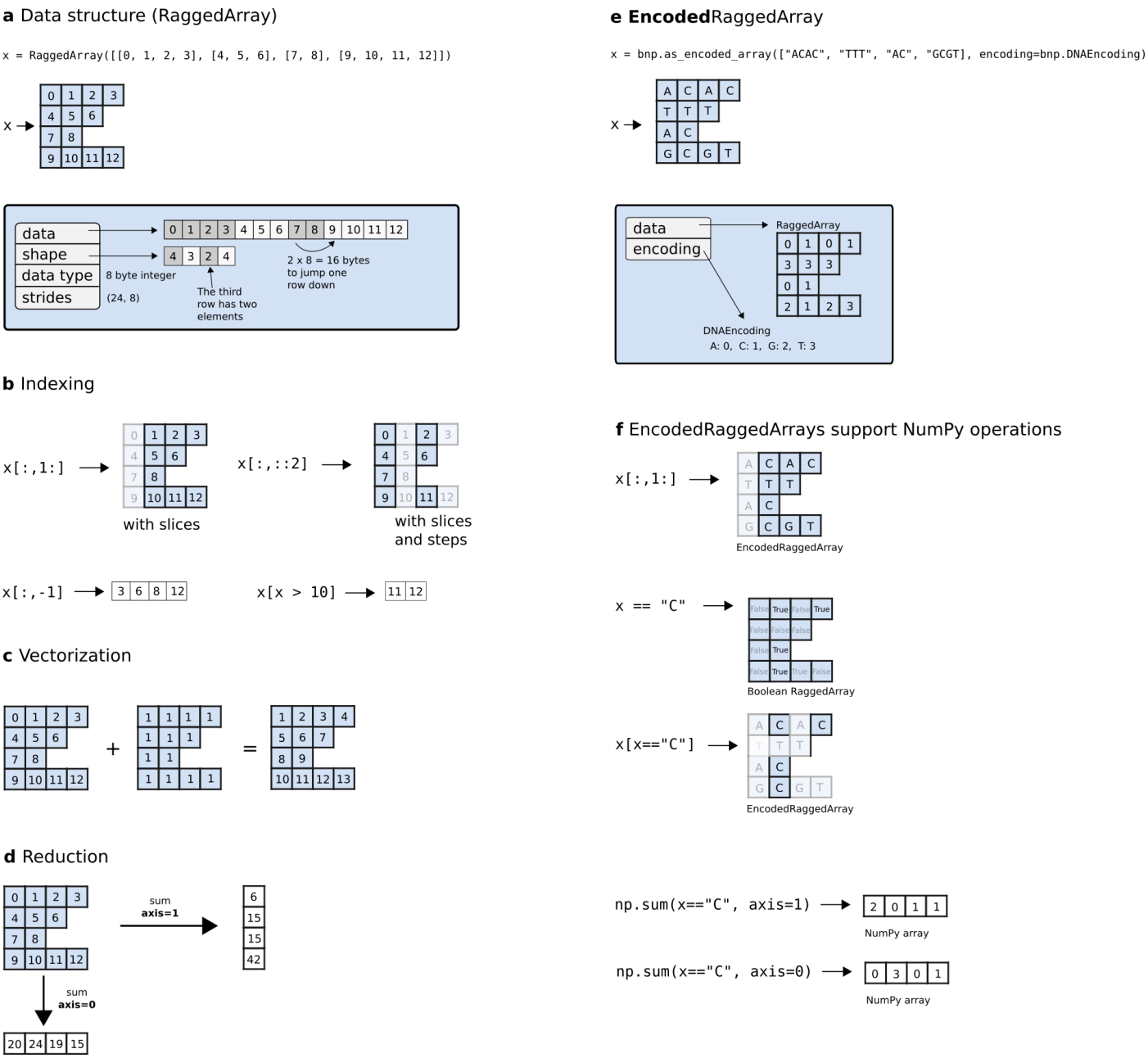


Figure 2: Overview of the RaggedArray and EncodedRaggedArray data structures. A RaggedArray is similar to a NumPy array/matrix but can represent a matrix consisting of rows with varying lengths (a). This makes it able to represent data with varying lengths efficiently in a shared data structure. A RaggedArray supports many of the same operations as NumPy arrays, such as indexing (b), vectorization (c) and reduction (d). These are implemented solely using NumPy, relying on functions like ufunc.accumulate, ufunc.reduceat and indexing. This means that most operations are close to equivalent operations on NumPy matrices, with a few exceptions like column reductions. An EncodedRaggedArray is a RaggedArray that supports storing and operating on non-numeric data (e.g. DNA sequences) by encoding the data and keeping track of the encoding (e). An EncodedRaggedArray supports the same operations as RaggedArrays (f). This figure is an adopted and modified version of Figure 1 in [5] and is licensed under a Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/).

BioNumPy has been developed following the principles of continuous integration and distribution. The codebase is thoroughly and automatically tested through an extensive collection of unit tests, application tests, integration tests and property-based tests [6]. New code changes are automatically benchmarked and tested before being automatically published, ensuring that updates can be frequent while high code quality is maintained. This workflow makes it safe and easy to allow

contributions from new contributors, which is important for longevity and community adoption of the package.

References

1. **BEDTools: a flexible suite of utilities for comparing genomic features**
   Aaron R Quinlan, Ira M Hall
   Bioinformatics (2010-01-28) https://doi.org/cmrms3
   DOI: 10.1093/bioinformatics/btq033 · PMID: 20110278 · PMCID: PMC2832824

2. **A fast, lock-free approach for efficient parallel counting of occurrences of k-mers**
   Guillaume Marçais, Carl Kingsford
   Bioinformatics (2011-01-07) https://doi.org/b7gkd6
   DOI: 10.1093/bioinformatics/btr011 · PMID: 21217122 · PMCID: PMC3051319

3. **Twelve years of SAMtools and BCFtools**
   Petr Danecek, James K Bonfield, Jennifer Liddle, John Marshall, Valeriu Ohan, Martin O Pollard, Andrew Whitwham, Thomas Keane, Shane A McCarthy, Robert M Davies, Heng Li
   GigaScience (2021-01-29) https://doi.org/gjxzc9
   DOI: 10.1093/gigascience/giab008 · PMID: 33590861 · PMCID: PMC7931819

4. **Benchmarking of deep neural networks for predicting personal gene expression from DNA sequence highlights shortcomings**
   Alexander Sasse, Bernard Ng, Anna E Spiro, Shinya Tasaki, David A Bennett, Christopher Gaiteri, Philip L De Jager, Maria Chikina, Sara Mostafavi
   Nature Genetics (2023-11-30) https://doi.org/gs7569
   DOI: 10.1038/s41588-023-01524-6 · PMID: 38036778

5. **Array programming with NumPy**
   Charles R Harris, KJarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, … Travis E Oliphant
   Nature (2020-09-16) https://doi.org/ghbzf2
   DOI: 10.1038/s41586-020-2649-2 · PMID: 32939066 · PMCID: PMC7759461

6. **Hypothesis: A new approach to property-based testing**
   David MacIver, Zac Hatfield-Dodds, Many Contributors
   Journal of Open Source Software (2019-11-21) https://doi.org/gs757j
   DOI: 10.21105/joss.01891