The Journal of Open Source Software

PyKronecker: A Python Library for the Efficient Manipulation of Kronecker Products and Related Structures

Edward Antonian[1], **Gareth W. Peters**[2], **and Michael Chantler**[1]

1 Heriot-Watt University, United Kingdom **2** University of California Santa Barbara, United States of America

DOI: 10.21105/joss.04900

Software

- Review
- Repository
- Archive

Editor: Kevin M. Moerman

Reviewers:

- @JulianKarlBauer
- @nicoguaro

Submitted: 07 October 2022
Published: 30 January 2023

Summary

Matrix operators composed of Kronecker products and related objects, such as Kronecker sums, arise in many areas of applied mathematics including signal processing, semidefinite programming, and quantum computing (Loan, 2000). As such, a computational toolkit for manipulating Kronecker-based systems, in a way that is both efficient and idiomatic, has the potential to aid research in many fields. PyKronecker aims to deliver this in the Python programming language by providing a simple API that integrates well with the widely-used NumPy library (Harris et al., 2020), and that supports automatic differentiation and accelerated computation on GPU/TPU hardware using Jax (Bradbury et al., 2018).

Kronecker products

The Kronecker product of an $(n \times n)$ matrix $\mathbf{A}$ and an $(m \times m)$ matrix $\mathbf{B}$, denoted $\mathbf{A} \otimes \mathbf{B}$, is defined by

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} \mathbf{A}_{1,1}\mathbf{B} & \dots & \mathbf{A}_{1,n}\mathbf{B} \\ \vdots & \ddots & \vdots \\ \mathbf{A}_{n,1}\mathbf{B} & \dots & \mathbf{A}_{n,n}\mathbf{B} \end{bmatrix}.$$

The resultant operator has shape $(nm \times nm)$ and, as such, can act on vectors of length nm. The Kronecker sum of $\mathbf{A}$ and $\mathbf{B}$, denoted $\mathbf{A} \oplus \mathbf{B}$ can be defined in terms of the Kronecker product as

$$\mathbf{A} \oplus \mathbf{B} = \mathbf{A} \otimes \mathbf{I}_m + \mathbf{I}_n \otimes \mathbf{B},$$

where $\mathbf{I}_d$ is the d-dimensional identity matrix, resulting in an operator of the same size as $\mathbf{A} \otimes \mathbf{B}$. By applying these definitions recursively, the Kronecker product or sum of more than two matrices can also be defined. In general, the Kronecker product/sum of k square matrices $\{\mathbf{A}^{(i)}\}_{i=1}^{k}$, with shapes $\{n_i \times n_i\}_{i=1}^{k}$ can be written respectively as

$$\bigotimes_{i=1}^{k} \mathbf{A}^{(i)} = \mathbf{A}^{(1)} \otimes \mathbf{A}^{(2)} \otimes \cdots \otimes \mathbf{A}^{(k)}$$

and

$$\bigoplus_{i=1}^{k} \mathbf{A}^{(i)} = \mathbf{A}^{(1)} \oplus \mathbf{A}^{(2)} \oplus \cdots \oplus \mathbf{A}^{(k)}.$$

The resultant operators can act on either vectors of length $N = \prod_{i=1}^{k} n_i$, or equivalently tensors of shape $(n_1, n_2, \dots n_k)$.

Efficient implementation of Kronecker-Vector Multiplication

Whilst a naive implementation of matrix-vector multiplication in this space has time and memory complexity of $O(N^2)$ a much more efficient implementation can be achieved. Work on this topic can be traced back to Roth (1934), however the first direct treatment can be found in Pereyra & Scherer (1973) and Boor (1979), both of which describe an efficient algorithm for the multiplication of a Kronecker product matrix onto a vector/tensor in algebraic terms. Later work such as Davio (1981), Buis & Dyksen (1996) and Fackler (2019) focused on optimising this algorithm further by considering other practical issues such as available hardware and physical memory layout. In particular, Fackler (2019) proposes the *kronx* algorithm, which forms the basis for the implementation found in PyKronecker, with some differences resulting from the C-style row-major memory layout used in Python as opposed to the Fortran-style column-major layout of Matlab, which was the target language of the aforementioned paper. In practice, by applying the kronx algorithm, the required memory and time complexity is reduced to $O(N)$ and $O(N \sum_{i=1}^{k} n_i)$ respectively. This makes it possible to solve many problems that would otherwise be intractable.

Statement of need

PyKronecker is aimed at researchers in any area of applied mathematics where systems involving Kronecker products arise. It has been designed with the following specific goals in mind.

a) *To provide a simple and intuitive object-oriented interface for manipulating systems involving Kronecker-products with Python.*

In PyKronecker, expressions are written in terms of a high-level operator abstraction. Users can define new composite operators by applying familiar matrix operations such as scaling, matrix addition/multiplication and transposition. This allows Kronecker operators to be manipulated as if they are large NumPy arrays, removing the need to write efficient but sometimes cryptic expressions involving the individual sub-matrices. This can greatly simplify code, making it easier to read, debug and refactor, allowing users to focus on their research goals without concerning themselves with underlying performance.

b) *To execute matrix-vector multiplications in a way that is maximally efficient and runs on parallel GPU/TPU hardware.*

Significant effort has gone into optimising the execution of matrix-vector and matrix-tensor multiplications. In particular, this comprises the kronx algorithm, Just In Time (JIT) compilation, and parallel processing on GPU/TPU hardware. As a result of this, PyKronecker is able to achieve very fast execution times compared to alternative implementations (see Table 1) .

c) *To allow automatic differentiation for complex loss functions involving Kronecker products.*

Many widely-used optimisation algorithms in Machine Learning (ML), such as stochastic gradient descent, rely on rapidly evaluating the derivative of an objective function. Automatic differentiation has played a key role in accelerating ML research by removing the need to manually derive analytical gradients (Baydin et al., 2018). By integrating with the Jax library, PyKronecker enables automatic differentiation of complex functions involving Kronecker products out of the box.

To the best of the our knowledge, no existing software achieves all three of these aims.

Comparison with existing libraries

One potential alternative in Python is the PyLops library which provides an interface for general functionally-defined linear operators, and includes a Kronecker product implementation (Ravasi & Vasconcelos, 2020). It also supports GPU acceleration with CuPy (Okuta et al., 2017). However, as a more general library, PyLops does not provide support for the Kronecker

product of more than two matrices, implement a Kronecker sum operator, implement matrix-tensor multiplication, or provide automatic differentiation. It is also significantly slower than PyKronecker when operating on simple NumPy arrays.

Another alternative is the library Kronecker.jl (Stock et al., 2020), implemented in the Julia programming language (Bezanson et al., 2017). Kronecker.jl has many of the same aims as PyKronecker and has a a clean interface, making use of Julia's support for unicode and infix functions to create Kronecker products with a custom $\otimes$ operator. However, at this time, the library does not support GPU acceleration or automatic differentiation, although the former is in development.

Table 1. shows a feature comparison of these libraries, along with the kronx algorithm implemented in "vanilla" (i.e., running on the CPU without JIT compilation) NumPy. The table also shows the time to compute the multiplication of a Kronecker product against a vector in two scenarios. In the first scenario, the Kronecker product is constructed from two matrices of size (400×400) and (500×500), and in the second scenario Kronecker product is constructed from three matrices of size (100×100), (150×150) and (200×200), respectively. Experiments were performed with an Intel Core 2.80GHz i7-7700HQ CPU, and an Nvidia 1050Ti GPU. In both cases, PyKronecker on the GPU is the fastest by a significant margin.

Implementation	Python	Autodiff	GPU support	Compute time (400, 500)	Compute time (100, 150, 200)
NumPy	Yes	No	No	5.04 ms $\pm$ 343 µs	38.9 ms $\pm$ 4.07 ms
Kronecker.jl	No	No	No	9.61 ms $\pm$ 881 µs	380 ms $\pm$ 6.15 ms
PyLops (CPU)	Yes	No	No	17.9 ms $\pm$ 986 µs	478 ms $\pm$ 4.79 ms
PyLops (GPU)	Yes	No	Yes	54.6 ms $\pm$ 1.04 ms	4.06 s $\pm$ 182 ms
PyKronecker (CPU)	Yes	Yes	No	1.92 ms $\pm$ 136 µs	15.1 ms $\pm$ 2.24 ms
PyKronecker (GPU)	Yes	Yes	Yes	261 µs $\pm$ 17.3 µs	220 µs $\pm$ 59.5 µs

Outlook and Future Work

There are several features that we are developing to expand the functionality of PyKronecker. The first is to provide support for non-square operators. In a typical problem, the Kronecker operators encountered represent simple linear transformations which preserve dimensionality. However, there are a significant minority of contexts where this is not the case. The inclusion of this feature would increase the range of possible applications. Secondly, we would like to add support for sparse matrices. This would enable computation with larger matrices and faster execution times where applicable. However this would require integration with Jax's sparse module, which is currently under development. Finally, for convenience, it may be useful to add some commonly used algorithms such as the conjugate gradient method for solving linear systems (Shewchuk, 1994), least squares, and various matrix decompositions such as eigenvalue, Cholesky and LU.

Acknowledgements

Thank you to Chris Krapu for valuable discussions, especially with regard to integrating PyKronecker with Jax, and automatic differentiation.

References

Baydin, A. G., Pearlmutter, B. A., Radul, A. A., & Siskind, J. M. (2018). Automatic differentiation in machine learning: A survey. *Journal of Marchine Learning Research*, *18*, 1–43. https://doi.org/10.48550/arXiv.1502.05767

Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. B. (2017). Julia: A fresh approach to numerical computing. *SIAM Review*, *59*(1), 65–98. https://doi.org/10.1137/141000671

Boor, C. D. (1979). Efficient computer manipulation of tensor products. *ACM Transactions on Mathematmal Software*, *5*, 173–182. https://doi.org/10.1145/355826.355831

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., & Zhang, Q. (2018). *JAX: Composable transformations of Python+NumPy programs* (Version 0.3.13) [Computer software]. http://github.com/google/jax

Buis, P. E., & Dyksen, W. R. (1996). Efficient vector and parallel manipulation tensor products. *ACM Transactions on Mathematical Software*. https://doi.org/10.1145/225545.225548

Davio, M. (1981). Kronecker products and shuffle algebra. *IEEE Transactions on Computers*, *C-30*, 116–125. https://doi.org/10.1109/TC.1981.6312174

Fackler, P. L. (2019). Algorithm 993: Efficient computation with Kronecker products. *ACM Trans. Math. Softw.*, *45*(2). https://doi.org/10.1145/3291041

Harris, C. R., Millman, K. J., Walt, S. J. van der, Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., Kerkwijk, M. H. van, Brett, M., Haldane, A., Río, J. F. del, Wiebe, M., Peterson, P., … Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, *585*(7825), 357–362. https://doi.org/10.1038/s41586-020-2649-2

Loan, C. F. V. (2000). The ubiquitous Kronecker product. *Journal of Computational and Applied Mathematics*, *123*(1), 85–100. https://doi.org/10.1016/S0377-0427(00)00393-9

Okuta, R., Unno, Y., Nishino, D., Hido, S., & Loomis, C. (2017). CuPy: A NumPy-compatible library for NVIDIA GPU calculations. *Proceedings of Workshop on Machine Learning Systems (LearningSys) in the Thirty-First Annual Conference on Neural Information Processing Systems (NIPS)*. http://learningsys.org/nips17/assets/papers/paper_16.pdf

Pereyra, V., & Scherer, G. (1973). *Efficient computer manipulation of tensor products with applications to multidimensional approximation* (Vol. 27). https://doi.org/10.1090/s0025-5718-1973-0395196-6

Ravasi, M., & Vasconcelos, I. (2020). PyLops—a linear-operator python library for scalable algebra and optimization. *SoftwareX*, *11*, 100361. https://doi.org/10.1016/j.softx.2019.100361

Roth, W. E. (1934). On direct product matrices. *Bulletin of the American Mathematical Society*. https://doi.org/10.2307/3609497

Shewchuk, J. R. (1994). *An introduction to the conjugate gradient method without the agonizing pain*. https://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf

Stock, M., Pahikkala, T., Airola, A., & Baets, B. D. (2020). *A general-purpose toolbox for efficient Kronecker-based learning*. https://doi.org/10.21105/jcon.00015