




JAXbind: Bind any function to JAX

Jakob Roth^{1,2,3*}, Martin Reinecke^{1*}, and Gordian Edenhofer^{1,2,4*}

¹ Max Planck Institute for Astrophysics, Karl-Schwarzschild-Str. 1, 85748 Garching, Germany ² Ludwig Maximilian University of Munich, Geschwister-Scholl-Platz 1, 80539 Munich, Germany ³ Technical University of Munich, Boltzmannstr. 3, 85748 Garching, Germany ⁴ Department of Astrophysics, University of Vienna, Türkenschanzstr. 17, A-1180 Vienna, Austria * These authors contributed equally.

DOI: [10.xxxxxx/draft](https://doi.org/10.xxxxxx/draft)

Software

- [Review](#) 
- [Repository](#) 
- [Archive](#) 

Editor: 

Submitted: 13 March 2024

Published: unpublished

License

Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License ([CC BY 4.0](https://creativecommons.org/licenses/by/4.0/)).

Summary

JAX is widely used in machine learning and scientific computing, the latter of which often relies on existing high-performance code that we would ideally like to incorporate into JAX. Reimplementing the existing code in JAX is often impractical and the existing interface in JAX for binding custom code requires deep knowledge of JAX and its C++ backend. The goal of JAXbind is to drastically reduce the effort required to bind custom functions implemented in other programming languages to JAX. Specifically, JAXbind provides an easy-to-use Python interface for defining custom so-called JAX primitives that support arbitrary JAX transformations.

Statement of Need

The use of JAX ([Bradbury et al., 2018](#)) is widespread in the natural sciences. Of particular interest is JAX's powerful transformation system. It enables to retrieve arbitrary derivatives of functions, batch computations, and just-in-time compile code for additional performance. Its transformation system relies on all components of the computation being written in JAX.

A plethora of high-performance code is not written in JAX and thus not accessible from within JAX. Rewriting these is often infeasible and/or inefficient. Ideally, we would like to mix existing high-performance code with JAX code. However, connecting code to JAX requires knowledge of the internals of JAX and its C++ backend.

In this paper, we present JAXbind, a package for bridging any function to JAX without in-depth knowledge of JAX's transformation system. The interface is accessible from Python without requiring any development in C++. The package is able to register any function, its partial derivatives and their transpose functions as a JAX native call, a so-called primitive.

We believe JAXbind to be highly useful in scientific computing. We intend to use this package to connect the Hartley transform and the spherical harmonic transform from DUCC ([Reinecke, 2024](#)) to the probabilistic programming package NIFTy ([Edenhofer et al., 2024](#)) as well as the radio interferometry response from DUCC with the radio astronomy package resolve ([Arras et al., 2024](#)). Furthermore, we intend to connect the non-uniform FFT from DUCC with JAX for applications in strong-lensing astrophysics. We envision many further applications within and outside of astrophysics.

To the best of our knowledge no other code currently exists for connecting generic functions to JAX. The package that comes the closest is Enzyme-JAX ([EnzymeAD, 2024](#)). Enzyme-JAX allows one to differentiate a C++ function with Enzyme ([W. S. Moses et al., 2021, 2022; W. Moses & Churavy, 2020](#)) and connect it together with its derivative to JAX. However, it enforces the use of Enzyme for deriving derivatives and does not allow for connecting arbitrary

41 code to JAX.

42 Automatic Differentiation and Code Example

43 Automatic differentiation is a core feature of JAX and often one of the main reasons for
 44 using it. Thus, it is essential that custom functions registered with JAX support automatic
 45 differentiation. In the following, we will outline which functions our package respectively JAX
 46 requires to enable automatic differentiation. For simplicity, we assume that we want to connect
 47 the nonlinear function $f(x_1, x_2) = x_1 x_2^2$ to JAX. The JAXbind package expects the Python
 48 function for f to take three positional arguments. The first argument, out, is a tuple into
 49 which the results are written. The second argument is also a tuple containing the input to
 50 the function, in our case, x_1 and x_2 . Via kwargs_dump, any keyword arguments given to the
 51 registered JAX primitive can be forwarded to f in a serialized form.

```
import jaxbind

def f(out, args, kwargs_dump):
    kwargs = jaxbind.load_kwargs(kwargs_dump)
    x1, x2 = args
    out[0][()] = x1 * x2**2
```

52 JAX's automatic differentiation engine can compute the Jacobian-vector product jvp and vector-
 53 Jacobian product vjp of JAX primitives. The Jacobian-vector product in JAX is a function
 54 applying the Jacobian of f at a position x to a tangent vector. In mathematical nomenclature
 55 this operation is called the pushforward of f and can be denoted as $\partial f(x) : T_x X \mapsto T_{f(x)} Y$,
 56 with $T_x X$ and $T_{f(x)} Y$ being the tangent spaces of X and Y at the positions x and $f(x)$. As the
 57 implementation of f is not JAX native, JAX cannot automatically compute the jvp. Instead,
 58 an implementation of the pushforward has to be provided, which JAXbind will register as the
 59 jvp of the JAX primitive of f . For our example, this Jacobian-vector-product function is given
 60 by $\partial f(x_1, x_2)(dx_1, dx_2) = x_2^2 dx_1 + 2x_1 x_2 dx_2$.

```
def f_jvp(out, args, kwargs_dump):
    kwargs = jaxbind.load_kwargs(kwargs_dump)
    x1, x2, dx1, dx2 = args
    out[0][()] = x2**2 * dx1 + 2 * x1 * x2 * dx2
```

61 The vector-Jacobian product vjp in JAX is the linear transpose of the Jacobian-vector product.
 62 In mathematical nomenclature this is the pullback $(\partial f(x))^T : T_{f(x)} Y \mapsto T_x X$ of f . Analogously
 63 to the jvp, the user has to implement this function as JAX cannot automatically construct it. For
 64 our example function, the vector-Jacobian product is $(\partial f(x_1, x_2))^T(dy) = (x_2^2 dy, 2x_1 x_2 dy)$.

```
def f_vjp(out, args, kwargs_dump):
    kwargs = jaxbind.load_kwargs(kwargs_dump)
    x1, x2, dy = args
    out[0][()] = x2**2 * dy
    out[1][()] = 2 * x1 * x2 * dy
```

65 To just-in-time compile the function, JAX needs to abstractly evaluate the code, i.e. it needs
 66 to be able to infer the shape and dtype of the output of the function given only the shape
 67 and dtype of the input. We have to provide these abstract evaluation functions returning the
 68 output shape and dtype given an input shape and dtype for f as well as for the vjp application.
 69 The output shape of the jvp is identical to the output shape of f itself and does not need
 70 to be specified again. The abstract evaluation functions take normal positional and keyword
 71 arguments.

```
def f_abstract(*args, **kwargs):
    assert args[0].shape == args[1].shape
```

```
return ((args[0].shape, args[0].dtype),)
```

```
def f_abstract_T(*args, **kwargs):
    return (
        (args[0].shape, args[0].dtype),
        (args[0].shape, args[0].dtype),
    )
```

72 We have now defined all ingredients necessary to register a JAX primitive for our function f
 73 using the JAXbind package.

```
f_jax = jaxbind.get_nonlinear_call(
    f, (f_jvp, f_vjp), f_abstract, f_abstract_T
)
```

74 f_jax is a JAX primitive registered via the JAXbind package supporting all JAX transformations.
 75 We can now compute the jvp and vjp of the new JAX primitive and even jit-compile and
 76 batch it.

```
import jax
import jax.numpy as jnp

inp = (jnp.full((4,3), 4.), jnp.full((4,3), 2.))
tan = (jnp.full((4,3), 1.), jnp.full((4,3), 1.))
res, res_tan = jax.jvp(f_jax, inp, tan)

cotan = (jnp.full((4,3), 6.),)
res, f_vjp = jax.vjp(f_jax, *inp)
res_cotan = f_vjp(cotan)

f_jax_jit = jax.jit(f_jax)
res = f_jax_jit(*inp)
```

77 Higher Order Derivatives and Linear Functions

78 JAX supports higher order derivatives and can differentiate a jvp or vjp with respect to the
 79 position at which the Jacobian was taken. Similar to first derivatives, JAX can not automatically
 80 compute higher derivatives of a general function f that is not natively implemented in JAX.
 81 Higher order derivatives would again need to be provided by the user. For many algorithms,
 82 first derivatives are sufficient, and higher order derivatives are often not implemented by high-
 83 performance codes. Therefore, the current interface of JAXbind is, for simplicity, restricted
 84 to first derivatives. In the future, the interface could be easily expanded if specific use cases
 85 require higher order derivatives.

86 In scientific computing, linear functions such as, e.g., spherical harmonic transforms are
 87 widespread. If the function f is linear, differentiation becomes trivial. Specifically for a linear
 88 function f , the pushforward respectively the jvp of f is identical to f itself and independent
 89 of the position at which it is computed. Expressed in formulas, $\partial f(x)(dx) = f(dx)$ if f is
 90 linear in x . Analogously, the pullback respectively the vjp becomes independent of the initial
 91 position and is given by the linear transpose of f , thus $(\partial f(x))^T(dy) = f^T(dy)$. Also, all
 92 higher order derivatives can be expressed in terms of f and its transpose. To make use of these
 93 simplifications, JAXbind provides a special interface for linear functions, supporting higher
 94 order derivatives, only requiring an implementation of the function and its transpose.

Platforms

Currently, JAXbind only supports primitives that act on CPU memory. In the future, GPU support could be added analogous to the CPU backend. The automatic differentiation in JAX is backend agnostic and would thus not require any additional bindings to work on the GPU.

Acknowledgements

We would like to thank Dan Foreman-Mackey for his detailed guide (<https://dfm.io/posts/extending-jax/>) on connecting C++ code to JAX. Jakob Roth acknowledges financial support from the German Federal Ministry of Education and Research (BMBF) under grant 05A23WO1 (Verbundprojekt D-MeerKAT III). Gordian Edenhofer acknowledges support from the German Academic Scholarship Foundation in the form of a PhD scholarship (“Promotionsstipendium der Studienstiftung des Deutschen Volkes”).

References

- Arras, P., Roth, J., Ding, S., Reinecke, M., Fuchs, R., & Johnson, V. (2024). *RESOLVE*. <https://gitlab.mpcdf.mpg.de/ift/resolve>
- 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). <http://github.com/google/jax>
- Edenhofer, G., Frank, P., Roth, J., Leike, R. H., Guerdi, M., Scheel-Platz, L. I., Guardiani, M., Eberle, V., Westerkamp, M., & Enßlin, T. A. (2024). *Re-Envisioning Numerical Information Field Theory (NIFTy.re): A Library for Gaussian Processes and Variational Inference*. <https://doi.org/10.48550/arXiv.2402.16683>
- EnzymeAD. (2024). *Enzyme-JAX* (Version 0.0.6). <https://github.com/EnzymeAD/Enzyme-JAX>
- Moses, W. S., Churavy, V., Paehler, L., Hückelheim, J., Narayanan, S. H. K., Schanen, M., & Doerfert, J. (2021). Reverse-mode automatic differentiation and optimization of GPU kernels via enzyme. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. <https://doi.org/10.1145/3458817.3476165>
- Moses, W. S., Narayanan, S. H. K., Paehler, L., Churavy, V., Schanen, M., Hückelheim, J., Doerfert, J., & Hovland, P. (2022). Scalable automatic differentiation of multiple parallel paradigms through compiler augmentation. *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. <https://doi.org/10.1109/SC41404.2022.00065>
- Moses, W., & Churavy, V. (2020). Instead of rewriting foreign code for machine learning, automatically synthesize fast gradients. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, & H. Lin (Eds.), *Advances in neural information processing systems* (Vol. 33, pp. 12472–12485). Curran Associates, Inc. <https://proceedings.neurips.cc/paper/2020/file/9332c513ef44b682e9347822c2e457ac-Paper.pdf>
- Reinecke, M. (2024). *DUCC: Distinctly useful code collection* (Version 0.33.0). <https://gitlab.mpcdf.mpg.de/mtr/ducc>