JAX-XC: EXCHANGE CORRELATION FUNCTIONALS LIBRARY IN JAX

Kunhao Zheng
Sea AI Lab
zhengkunhao@sea.com

Min Lin
Sea AI Lab
linmin@sea.com

ABSTRACT

We present JAX-XC an open-source library that provides exchange-correlation functionals in Jax. JAX-XC is built from LIBXC, its correctness has been verified numerically against LIBXC. Thanks to Jax, JAX-XC is end-to-end differentiable, computationally more efficient thanks to the vectorization provided by XLA, and also portable on various accelerators. More importantly, as more research is focusing on machine learning for density functional theory, we hope that JAX-XC could serve as a deep learning-friendly tool and a stepping-stone for researchers working in the intersection of deep learning and density functional theory.

1 INTRODUCTION

With its predominant application on molecular and material, Kohn-Sham (KS) Density Functional Theory (DFT) is the most widely-used method in computational quantum chemistry. Exchange-correlation (XC) functional is at the core of KS-DFT: it includes information related to quantum exchange and correlation effects. The exact XC functional can not be written analytically, and the choice of different approximations is the key to the accuracy of the computation.

Researchers propose many explicit forms as approximations of the exact XC functional for practical numerical implementation. Efforts have been made for decades; among them, there are famous functionals like B88 (Becke, 1988), PW91 (Perdew et al., 1992), and PBE (Perdew et al., 1996) proposed with parameters fit to experimental data. At the time of writing, over 700 XC functionals are implemented in LIBXC (Marques et al., 2012; Lehtola et al., 2018), a software libary that collects and numerically implements XC functionals. Implementations in LIBXC are ubiquitously used in downstream software packages. Examples include but do not limit to PySCF (Sun et al., 2018; 2020), Psi4 (Smith et al., 2020) and BigDFT (Ratcliff et al., 2020).

Recently, in the vein of AI for science, DFT has attracted increasing attention from deep learning (DL) community (Chen et al., 2020; Kalita et al., 2021; Li et al., 2023). Many research efforts have been paid to learn either a black-box (Lei & Medford, 2019; Dick & Fernandez-Serra, 2020; Nagai et al., 2020; Dick & Fernandez-Serra, 2021; Kasim & Vinko, 2021; Ryabov et al., 2020; Kirkpatrick et al., 2021) or an analytical-form (Ma et al., 2022; Kovács et al., 2022) XC functional, in which existing XC functionals serves as a sub-computation or as the baseline for benchmarking. In these existing works, the researchers need to integrate DFT software and deep learning frameworks, for example, LIBXC is not particularly targeting the deep learning community. It is implemented in C, and although it does provide high-order derivatives of the functionals, it is still not straightforward to integrate with deep learning frameworks. This hinders end-to-end differentiability when deep learning-based methods are explored for DFT. The same limitation applies to other quantum chemistry libraries e.g. LIBCINT for integral computation. Extra efforts are needed to bridge this gap.

To date, the two communities (DL & DFT) have not organized their efforts around toolings towards a fully-unified paradigm to spur the research advance under deep learning framework. To address this need and to provide a common resource for research groups working on the intersection of DL and DFT, we present JAX-XC[1], a flexible, end-to-end-differentiable and open-source library that

[1]https://github.com/sail-sg/jax_xc

```python
import jax
import jax.numpy as jnp
import jax.scipy.stats.norm as norm
import pylibxc

# a 3D gaussian density function
def rho(r):
  return jnp.prod(norm.pdf(r))

# a grid point in 3D
r = jnp.array([0.1, 0.2, 0.3])

func = pylibxc.LibXCFunctional("gga_c_pbe", 1)

# manually construct derivative function
d_rho = jax.grad(rho)
sigma = lambda r: jnp.dot(d_rho(r), d_rho(r))
d_sigma = jax.grad(sigma)

# manually prepare sigma
inp = {"rho": rho(r), "sigma": sigma(r)}
res = func.compute(inp)

# get exc and its 1st order derivative
exc_r = res["zk"]
df_r = res["vrho"] * d_rho(r) + res["vsigma"] * d_sigma(r)
```

```python
import jax
import jax.numpy as jnp
import jax.scipy.stats.norm as norm
import jax_xc

# a 3D gaussian density function
def rho(r):
  return jnp.prod(norm.pdf(r))

# a grid point in 3D
r = jnp.array([0.1, 0.2, 0.3])

exc = jax_xc.gga_c_pbe(rho, polarized=False)
f = lambda r: rho(r) * exc(r)

# get exc and its 1st order derivative
exc_r = exc(r)
df_r = jax.grad(f)(r)
```

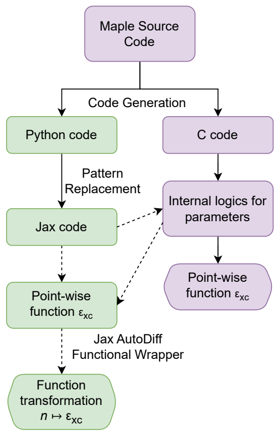
Maple Source Code
Code Generation
Python code
C code
Pattern Replacement
Internal logics for parameters
Jax code
Point-wise function ε_{xc}
Point-wise function ε_{xc}
Jax AutoDiff Functional Wrapper
Function transformation $n \mapsto \varepsilon_{xc}$

Figure 1: **Left**: Computation of gga_c_pbe XC functionals and its derivative in LIBXC API. **Middle**: Same computation in JAX-XC API. **Right**: Design of JAX-XC. Purple block is LIBXC module; green block is JAX-XC module. Solid lines are in build time; dotted lines are in runtime.

provides XC functionals in Jax[2] (Bradbury et al., 2018). JAX-XC is designed with the usecase of deep learning in mind, easy to use and seamlessly integrable in modern deep learning framework. JAX-XC leverages the powerful features in Jax to accelerate on various hardwares with calculation faster than LIBXC, and unlock differentiability up to unlimited order. We intend JAX-XC to be a deep learning-friendly tool in the field of AI for quantum chemistry and serve as a stepping-stone for researchers to explore and build upon.

2 JAX-XC

We first introduce XC functionals and the implementation of LIBXC. We then detail the design of JAX-XC, including the API and the engineering efforts.

2.1 XC FUNTIONALS AND IMPLEMENTATION OF LIBXC

XC functionals compute the XC energy given the electron density as input. It takes the general form of $E_{\text{xc}}[\rho] = \int \rho(\boldsymbol{r})\varepsilon_{\text{xc}}(\boldsymbol{r})d\boldsymbol{r}$, where $\boldsymbol{r}$ is a position in 3D space, ρ is the particle density and ε_{xc} is the XC energy density per unit particle. LIBXC stores and computes numerous forms of ε_{xc}. XC functionals are divided into the following categories: local density approximation (LDA), general-gradient approximation (GGA) and meta-GGA.

In LIBXC's C code, ε_{xc} is implemented in a point-wise evaluation manner: it takes several scalars (the evaluation of the density, etc.) as input and output a scalar. Implementation-wise, the differences among categories are the input arguments of ε_{xc}: for LDA, $\varepsilon_{\text{xc}}(\boldsymbol{r}) = \varepsilon_{\text{xc}}(\rho(\boldsymbol{r}))$; for GGA, $\varepsilon_{\text{xc}}(\boldsymbol{r}) = \varepsilon_{\text{xc}}(\rho(\boldsymbol{r}), |\nabla\rho(\boldsymbol{r})|^2)$; for Meta-GGA, $\varepsilon_{\text{xc}}(\boldsymbol{r}) = \varepsilon_{\text{xc}}(\rho(\boldsymbol{r}), |\nabla\rho(\boldsymbol{r})|^2, \nabla^2\rho(\boldsymbol{r}), \frac{1}{2}\sum|\nabla\psi(\boldsymbol{r})|^2)$. ψ denotes the molecular orbitals. Therefore, users need to prepare the input, e.g. evaluation of the density gradient at a certain position, during the computation. We omit the spin for presentation simplicity.

With the source code written in maple (Maplesoft, 2018), LIBXC leverages the code generation feature to obtain C code. This comes with several advantages. The maple source code is purely functional, more readable and closer to the mathematics formula presented in the corresponding paper. It is easier to compute the derivative of the functionals w.r.t. input arguments using maple's symbolic derivative. However, the available derivative is up to a certain order depending on the maple command in build time.

[2]Jax is a differentiable Python library for machine learning.

2.2 Library Design

We take a fresh depart from LIBXC's design. Instead of preparing the point-wise evaluations on $\rho(\boldsymbol{r})$, $\nabla\rho(\boldsymbol{r})$ etc, JAX-XC provides a functional API, where different types of functionals are unified under the same umbrella of function transformation. JAX-XC defines the function transformation $\hat{\mathcal{F}}$ which takes the density function ρ as input and output the function $\varepsilon_{\text{xc}} = \hat{\mathcal{F}}(\rho)$. Function transformation is natively implemented in Jax. This comes with several advantages over LIBXC. Users could evaluate ε_{xc} at a variable size of grid points to obtain the value $\varepsilon_{\text{xc}}(\boldsymbol{r})$, or get derivative of ε_{xc} up to unlimited order. We compare the code using LIBXC and JAX-XC in Figure 1.

These features are achieved by modifying the maple source code to generate Python code instead of C code. Thereafter, we use global pattern matching rules and wrappers to transform the Python code into Jax-compatible program. Finally, we use Jax's auto-differentiation feature to wrap the gradient operation inside the API for different types of functionals. The evaluation of the function is decoupled from the function transformation to enable a grid-point free feature.

2.3 Implementation Details

Handling Parameters A large number of functionals have hyper-parameters. Hyper-parameters customize the behavior of the functionals, e.g. switch between paramagnetism and ferromagnetism mode, or modify the coefficients used inside the functionals. Notably, LIBXC preprocesses the hyper-parameters before sending them to the main computation. While the main computation is implemented in maple and easily convertible to Python, the preprocessing code is implemented in C and differs for each functional. It is tedious to reimplement each of them in Python. Therefore, we modify the original LIBXC code to expose and share the preprocessing code with JAX-XC. The overall structure is shown in Figure 1(right). By reusing preprocessing code in LIBXC, we avoid heavy manual efforts of translation and maintenance.

Build System We use `bazel` for automating the build and test of JAX-XC. `bazel` automates the following steps, 1. clone LIBXC source code; 2. modify the source code to expose the preprocessing logics; 3. generate Python code from maple source code; 4. wrap the core computation with clean APIs and generate the documentation; 5. finally, package the library as a Python wheel.

Documentation[3] We provide information on the user customizable parameters for each functional. We also list the dependencies on other functionals when it comes to hybrid functionals. The contents of the documentation are generated from LIBXC's bibtex entries and parameter descriptions.

Coverage Our goal is to cover all functionals presented in LIBXC. However, there are still a dozen of functionals not supported in JAX-XC (Appendix B). We leave special treatement to these functionals as future work and we welcome contribution from the community.

License JAX-XC is released under MPL 2.0 license, aligned with LIBXC.

3 Experiments

3.1 Numerical Error

Generated from the same maple code, JAX-XC should theoretically perform exactly the same computations as LIBXC. However, they could fail to align due to human error. There are also many sources of non-determinism that could break bitwise equality in their results, i.e., the change of operation order due to maple CodeGen's non-determinism; the various optimizations performed by XLA. Therefore, it is crucial to test numerically that the difference between JAX-XC and LIBXC is within a tolerable range. With the same input values, we test the numerical differences of the outputs generated from JAX-XC and LIBXC. The input values (density, density gradient, etc.) are all randomly sampled from 10^{-5} to 10^{2}. Since the C code is compiled with `double` precision, we also enable `float64` in JAX to perform an apple-to-apple comparison.

[3]The documentation is hosted on `https://jax-xc.readthedocs.io/en/latest/`

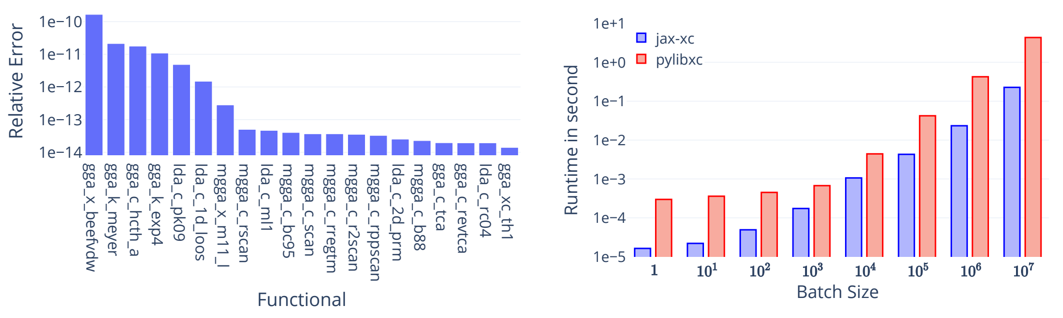

Figure 2: **Left**: Relative Error r between JAX-XC and LIBXC. The y-axis is log-scale. We present the 20 functionals with the largest r in descending order. **Right**: Runtime of JAX-XC and LIBXC across different batch size. The y-axis is log-scale.

We denote the output from LIBXC as y_1, the result from JAX-XC as y_2. The numerical error is tested based on both the absolute tolerance a and relative tolerance r. The criterion of passing the test is:

$$|y_2 - y_1| \leq a + r \cdot y_1. \tag{1}$$

There are 213 core routines shared by the 700 functionals (considering most of them are hybrid functionals). All of them pass the test for $a = r = 2 \times 10^{-10}$, among which $184/213$ of the core routines attains $a = r = 1 \times 10^{-14}$. We present the functionals with the highest relative errors in Figure 2 (left). We found that the order of multiplication and addition which maple generated differently for C and Python accounts for the high numerical error in the figure. In fact, Lehtola & Marques (2022) found a list of density functionals that are numerically ill-behaved, which overlaps with the functionals we found with high numerical error, including `gga_x_beefvdw`[4], `gga_k_meyer`, `lda_c_pk09` and `mgga_x_m11_l`.

3.2 SPEED BENCHMARK

We conduct our experiments on a 64-core machine with Intel(R) Xeon(R) Silver 4216 CPU @ 2.10GHz. For a fair comparison, the experiments of LIBXC and JAX-XC are in a CPU-only environment. We exclude the compilation time of JAX-XC and enable `float64`. We use the Python binding of LIBXC (PYLIBXC in Figure 2). In numerical integration ε_{xc} needs to be evaluated on a grid of coordinates, therefore, we also evaluate across the different batch sizes of inputs.

Figure 2 (right) shows that across all batch sizes from 1 to 10^7. JAX-XC runs constantly faster than PYLIBXC. The speed-up ranges from $\mathbf{3}\times$ to $\mathbf{10}\times$, with a higher speed-up for larger batch sizes. We hypothesize that when the batch size is large, the speed-up mainly comes from the vectorized backend of Jax. Other optimizations performed in the XLA compiler could also contribute to this advantage, e.g. instruction fusion, constant folding, etc. For small batch sizes from 1 to 10^3, we observe a nearly constant runtime in PYLIBXC. With some further profiling, it turns out that the main overhead comes from the dispatching code that invokes C from Python. This points to the possibility of further optimization in PYLIBXC's Python code.

4 CONCLUSION

We present JAX-XC, an end-to-end-differentiable XC functional library that translates existing XC functionals in LIBXC to Jax. We conduct experiments to validate the correctness of implementation. We also show a significant computation acceleration thanks to the vectorized backend of Jax. JAX-XC enables full-differentiability when integrating existing XC functionals with machine learning. We hope that JAX-XC could help accelerate the research at the junction of machine learning and DFT.

[4]In Appendix D, we provide a detailed analysis for `gga_x_beefvdw` (Wellendorff et al., 2012), the functional with the highest numerical error.

Acknowledgments

We thank Susi Lehtola for his answers to our questions concerning LIBXC. We thank Tianbo Li, Zekun Shi, Zheyuan Hu and Giovanni Vignale for their time for discussion and their encouragement along the development of JAX-XC. We also thank the infra team in Sea AI Lab for their support.

References

Axel D Becke. Density-functional exchange-energy approximation with correct asymptotic behavior. *Physical review A*, 38(6):3098, 1988.

Axel D Becke and Marc R Roussel. Exchange holes in inhomogeneous systems: A coordinate-space model. *Physical Review A*, 39(8):3761, 1989.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL `http://github.com/google/jax`.

Yixiao Chen, Linfeng Zhang, Han Wang, and Weinan E. Deepks: A comprehensive data-driven approach toward chemically accurate density functional theory. *Journal of Chemical Theory and Computation*, 17(1):170–181, 2020.

Sebastian Dick and Marivi Fernandez-Serra. Machine learning accurate exchange and correlation functionals of the electronic density. *Nature communications*, 11(1):3509, 2020.

Sebastian Dick and Marivi Fernandez-Serra. Using differentiable programming to obtain an energy and density-optimized exchange-correlation functional. *arXiv preprint arXiv:2106.04481*, 2021.

Bhupalee Kalita, Li Li, Ryan J McCarty, and Kieron Burke. Learning to approximate density functionals. *Accounts of Chemical Research*, 54(4):818–826, 2021.

M. F. Kasim and S. M. Vinko. Learning the exchange-correlation functional from nature with fully differentiable density functional theory. *Phys. Rev. Lett.*, 127:126403, Sep 2021.

James Kirkpatrick, Brendan McMorrow, David HP Turban, Alexander L Gaunt, James S Spencer, Alexander GDG Matthews, Annette Obika, Louis Thiry, Meire Fortunato, David Pfau, et al. Pushing the frontiers of density functionals by solving the fractional electron problem. *Science*, 374 (6573):1385–1389, 2021.

Péter Kovács, Fabien Tran, Peter Blaha, and Georg KH Madsen. What is the optimal mgga exchange functional for solids? *The Journal of Chemical Physics*, 157(9):094110, 2022.

Susi Lehtola and Miguel AL Marques. Many recent density functionals are numerically ill-behaved. *The Journal of Chemical Physics*, 157(17):174114, 2022.

Susi Lehtola, Conrad Steigemann, Micael JT Oliveira, and Miguel AL Marques. Recent developments in libxc—a comprehensive library of functionals for density functional theory. *SoftwareX*, 7:1–5, 2018.

Xiangyun Lei and Andrew J Medford. Design and analysis of machine learning exchange-correlation functionals via rotationally invariant convolutional descriptors. *Physical Review Materials*, 3(6):063801, 2019.

Tianbo Li, Min Lin, Zheyuan Hu, Kunhao Zheng, Giovanni Vignale, Kenji Kawaguchi, A.H. Castro Neto, Kostya S. Novoselov, and Shuicheng YAN. D4FT: A deep learning approach to kohn-sham density functional theory. In *International Conference on Learning Representations*, 2023. URL `https://openreview.net/forum?id=aBWnqqsuot7`.

He Ma, Arunachalam Narayanaswamy, Patrick F. Riley, and Li Li. Evolving symbolic density functionals. *Science Advances*, 8(36), 2022.

Maplesoft. Maple, 2018. URL `https://www.maplesoft.com`. Maplesoft, a division of Waterloo Maple Inc.

Miguel AL Marques, Micael JT Oliveira, and Tobias Burnus. Libxc: A library of exchange and correlation functionals for density functional theory. *Computer physics communications*, 183 (10):2272–2281, 2012.

Ryo Nagai, Ryosuke Akashi, and Osamu Sugino. Completing density functional theory by machine learning hidden messages from molecules. *npj Computational Materials*, 6(1):1–8, 2020.

John P Perdew, John A Chevary, Sy H Vosko, Koblar A Jackson, Mark R Pederson, Dig J Singh, and Carlos Fiolhais. Atoms, molecules, solids, and surfaces: Applications of the generalized gradient approximation for exchange and correlation. *Physical review B*, 46(11):6671, 1992.

John P Perdew, Kieron Burke, and Matthias Ernzerhof. Generalized gradient approximation made simple. *Physical review letters*, 77(18):3865, 1996.

Laura E. Ratcliff, William Dawson, Giuseppe Fisicaro, Damien Caliste, Stephan Mohr, Augustin Degomme, Brice Videau, Viviana Cristiglio, Martina Stella, Marco D'Alessandro, Stefan Goedecker, Takahito Nakajima, Thierry Deutsch, and Luigi Genovese. Flexibilities of wavelets as a computational basis set for large-scale electronic structure calculations. *The Journal of Chemical Physics*, 152(19):194110, 2020.

Alexander Ryabov, Iskander Akhatov, and Petr Zhilyaev. Neural network interpolation of exchange-correlation functional. *Scientific reports*, 10(1):1–7, 2020.

Daniel GA Smith, Lori A Burns, Andrew C Simmonett, Robert M Parrish, Matthew C Schieber, Raimondas Galvelis, Peter Kraus, Holger Kruse, Roberto Di Remigio, Asem Alenaizan, et al. Psi4 1.4: Open-source software for high-throughput quantum chemistry. *The Journal of chemical physics*, 152(18):184108, 2020.

Qiming Sun, Timothy C Berkelbach, Nick S Blunt, George H Booth, Sheng Guo, Zhendong Li, Junzi Liu, James D McClain, Elvira R Sayfutyarova, Sandeep Sharma, et al. Pyscf: the python-based simulations of chemistry framework. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 8(1):e1340, 2018.

Qiming Sun, Xing Zhang, Samragni Banerjee, Peng Bao, Marc Barbry, Nick S Blunt, Nikolay A Bogdanov, George H Booth, Jia Chen, Zhi-Hao Cui, et al. Recent developments in the pyscf program package. *The Journal of chemical physics*, 153(2):024109, 2020.

Jess Wellendorff, Keld T Lundgaard, Andreas Møgelhøj, Vivien Petzold, David D Landis, Jens K Nørskov, Thomas Bligaard, and Karsten W Jacobsen. Density functionals for surface science: Exchange-correlation model development with bayesian error estimation. *Physical Review B*, 85 (23):235149, 2012.

A LIBXC VERSION

JAX-XC is currently building upon LIBXC version 6.0.0: `https://gitlab.com/libxc/libxc/-/tree/6.0.0`

B NOT AVAILABLE FUNCTIONALS

We present the list of not available functionals in the list below. Some are due to technical difficulties. For example, the Becke-Roussel exchange functional (Becke & Roussel, 1989) does not have an closed analytic form and its numerical solution in LIBXC is implemented in C. Others are due to Jax's lack of support. For example, Jax is extremely slow when calling `exp1` function in batch[5], by which the functionals are affected could be supported once the issue is solved.

- Becke-Roussel functional (Becke & Roussel, 1989) not having a closed-form expression:

  ```
  gga_x_fd_lb94
  gga_x_fd_revlb94
  gga_x_gg99
  gga_x_kgg99
  hyb_gga_xc_case21
  hyb_mgga_xc_b94_hyb
  hyb_mgga_xc_br3p86
  mgga_c_b94
  mgga_x_b00
  mgga_x_bj06
  mgga_x_br89
  mgga_x_br89_1
  mgga_x_mbr
  mgga_x_mbrxc_bg
  mgga_x_mbrxh_bg
  mgga_x_mggac
  mgga_x_rpp09
  mgga_x_tb09
  ```

- Requiring explicit 1D integration:

  ```
  lda_x_1d_exponential
  lda_x_1d_soft
  ```

- JIT too long for `exp1`:

  ```
  gga_x_wpbeh
  gga_c_ft97
  ```

- vxc functional not comparable to LIBXC direct computation:

  ```
  lda_xc_tih
  gga_c_pbe_jrgx
  gga_x_lb
  ```

C COMPARISON OF RUNTIME RATIO ACROSS BATCH SIZE

We present the distribution of runtime ratio of JAX-XC and LIBXC in Figure 3, computed as the runtime of JAX-XC divided by the runtime of LIBXC. We exclude one datapoint `mgga_x_2d_prhg07` from the runtime ratio visualization because it is an outlier due to Jax's lack of support of `lamberw`

[5]`https://github.com/google/jax/issues/13543`

function[6] and we use `tensorflow_probability.substrates.jax.math.lambertw` instead. We observe a 3x-5x speed down in this functional. We note that this datapoint is only excluded in the visualization in Figure 3 but is taken into account in the calculation of average speedup in Figure 2.

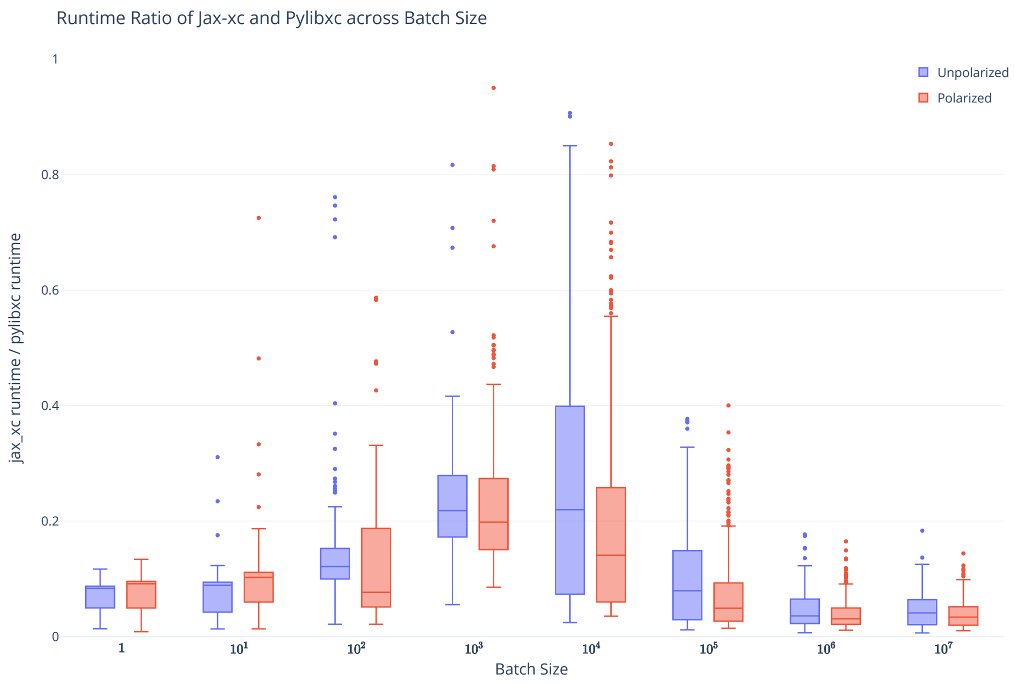

Figure 3: Distribution of runtime ratio across batch size.

D Source of Numerical Error in gga_x_beefvdw

gga_x_beefvdw is the functional with the largest numerical error when the outputs from JAX-XC and LIBXC are compared (Figure 2, left). It involves computing the sum of a series of Lengendre polynomial from order 0 to order 30, whose coefficients are obtained by fitting the experimental data (c.f. Table III in Wellendorff et al. (2012)).

Here we layout the analysis of the core program (unpolarized version) from both libraries, generated from the same maple source code but are displayed in Python and C respectively.

D.1 C Code in LIBXC

```
t2 = rho[0] / 0.2e1 <= p->dens_threshold;
t3 = M_CBRT3;
t4 = M_CBRTPI;
...
t78 = -0.69459735177638985466e0 * t45 + 0.52755620115589800943e0 * t47 -
0.38916037779196815969e0 * t44 - 0.44233229018433803622e3 * t50 -
0.61754786104528599731e3 * t52 + 0.37835396407252402359e4 * t54 -
0.72975787893717136018e1 * t56 + 0.30542034959315850168e2 * t58 +
0.86005730499279641299e2 * t60 - 0.54277774626371860 32e4 * t64 +
0.41355861880146538 75e4 * t63 * t66 - 0.29150193011493262292e5 * t70 +
0.40074935854432390114e5 * t72 + 0.90365611108522808258e5 * t74 -
0.16114215399846280595e6 * t76;
...
t102 = 0.11313514630621233134e1 - 0.13204466182182150467e6 * t79 +
```

[6] https://github.com/google/jax/issues/13680

```
0.2558947952623533461e6 * t81 - 0.32352403136049329184e6 * t83 +
0.18078200670879145336e6 * t85 - 0.12981481812794983922e6 * t87 +
0.56174007979372666951e5 * t89 + 0.27967048856303053872e6 * t91 -
0.10276426607863824397e5 * t93 - 0.16837084139014120539e6 * t63 -
0.281024018056846299e4 * t66 + 0.70504541869034010051e5 * t97 +
0.22748997850816485208e4 * t69 - 0.20148245175625047025e5 * t62 +
0.37534251004296526981e-1 * t41;
t103 = t78 + t102;
t107 = my_piecewise3(t2, 0, -0.3e1 / 0.8e1 * t6 * t19 * t103);
tzk0 = 0.2e1 * t107;
```

D.2 Python Code in jax-xc

```
t3 = jnp.cbrt(3)
...
t87 = 0.4135586188014653875e4 * t50 * t49 - 0.5427777462637186032e4 * t50 * t53 +
0.40074935854432390114e5 * t50 * t56 - 0.29150193011493262292e5 * t50 * t60 +
0.90365611108522808258e5 * t50 * t63 - 0.16114215399846280595e6 * t50 * t66 -
0.13204466182182150467e6 * t50 * t48 + 0.2558947952623533461e6 * t50 * t71 -
0.32352403136049329184e6 * t50 * t46 + 0.18078200670879145336e6 * t50 * t47 -
0.12981481812794983922e6 * t50 * t59 + 0.56174007979372666951e5 * t50 * t44 +
0.27967048856303053872e6 * t50 * t45 - 0.16837084139014120539e6 * t50 +
0.70504541869034010051e5 * t48 * t71
...
t103 = 0.11313514630621233134e1 - 0.10276426607863824397e5 * t48 * t66 -
0.281024018056846299e4 * t49 + 0.22748997850816485208e4 * t60 -
0.20148245175625047025e5 * t53 + 0.37835396407252402359e4 * t56 -
0.44233229018433803622e3 * t48 - 0.61754786104528599731e3 * t63 +
0.86005730499279641299e2 * t66 - 0.72975787893717136018e1 * t47 +
0.30542034959315850168e2 * t71 - 0.69459735177638985466 * t46 -
0.38916037779196815969 * t45 + 0.52755620115589800943 * t59 +
0.37534251004296526981e-1 * t42
t108 = jnp.where(r0 / 0.2e1 <= p.dens_threshold, 0,
  -0.3e1 / 0.8e1 * t3 / t4 * t18 * t19 * (t87 + t103))
tzk0 = 0.2e1 * t108
```

From the generated code, we could see that in both libraries, maple's code generation mechanism splits the sum of the series of Lengendre polynomials from order 0 to order 30 into 2 temporary variables (`t78` and `t102` in C code, `t87` and `t103` in Python code), which are later summed together. However, the behavior of splitting the summation into 2 groups is not consistent in Python and C.

For example, if we give the input of density $n = 1$ and the square norm of the density gradient $\sigma = |\nabla n \cdot \nabla n| = 1$, the 2 temporary variables in Python will be 4950.3740984881515 and -4949.336203207162, while the other 2 variables in C will be 99989.78580149758 and -99988.74790621664. The summation of them gives 1.0378952809896873 and 1.0378952809405746, where there is already a numerical error of order 10^{-11}.

Since both versions create temporary variables and do not fully reflect the analytic solution, more analysis on what implementation is closer to the analytic solution is needed.