The Journal of Open Source Software

BaderKit: A Python Package for Grid-based Bader Charge Analysis

Samuel M. Weaver [1*] **and Scott Warren** [1*¶]

1 University of North Carolina Chapel Hill, United States ¶ Corresponding author * These authors contributed equally.

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor: Bonan Zhu

Reviewers:

- @obaica
- @mdavezac

Submitted: 30 October 2025
Published: unpublished

Summary

The concept of oxidation states has existed for centuries, guiding and informing the decisions of generations of scientists. However, oxidation states are not observable, and cannot be uniquely derived from first principle calculations. This has led researchers to develop a variety of methods to recover oxidation states, each with their own unique theory and methodology. Chief among these methods is that described by Bader in his Quantum Theory of Atoms in Molecules, which derives oxidation states directly from a systems electron charge density. The `BaderKit` package brings Bader charge analysis into the modern Python ecosystem, and reworks the most popular grid-based algorithms to run in parallel.

Statement of Need

Bader Quantum Theory of Atoms in Molecules (QTAIM) charge analysis is among the most widely used methods in chemistry and materials science, with thousands of articles referencing the method each year. This popularity has given rise to many packages for performing QTAIM analysis. Despite the variety of implementations, none are well equipped for modern high-throughput workflows. Most are written in Fortran making them cumbersome to automate and adapt for purposes outside their original scope. The algorithms implemented in these codes are serial and do not utilize modern multi-core CPUs. `BaderKit` aims to resolve these issues, providing a fast, parallelized, and easily extended QTAIM implementation written in Python.

State of the Field

The most popular implementation is the `Bader v1.05` code developed by the Henkelman group at UT Austin (Henkelman et al., 2006) who pioneered the grid-based QTAIM alorithms. It is fast and memory efficient, but its use of Fortran means that workflows using the code must call it as a subprocess. As a result, much of the useful information generated by the process must be read from file or is lost entirely. Additionally, it is fully serial and does not utilize modern CPU's to their full extent. Another popular Fortran implementation, `Critic2`(Otero-de-la-Roza et al., 2014) provides additional methods and a convenient graphical interface, but suffers from similar automation issues. There have been some previous attempts to alleviate these issues. In particular, `pybader` (Kerrigan, 2020) partially implemented the method into parallelized Python code. However, `pybader` is typically slower than the other implementations and requires a significant amount of boilerplate code. `BaderKit` aims to improve upon the areas where each of these codes falls short. It is fast, extensive, easy to use, and designed to be easily inserted into modern workflows.

Software Design

The foremost goal of `BaderKit` is to make Bader charge analysis broadly accessible and easy to implement. It is available through `github`, `PyPi`, and `conda-forge` and runs on the most popular operating systems (e.g. Windows, MacOS, Ubuntu, etc.). The object-oriented API is based on the widely used `PyMatGen`(Ong et al., 2013) allowing users to obtain atomic charges with just three lines of code. `BaderKit` runs directly on the output most popular density functional theory codes (e.g. VASP (Kresse & Furthmüller, 1996), Gaussian (Frisch et al., 2016), Quantum Espresso (Giannozzi et al., 2009)), and can be easily extended to run on the output of others. These choices allow `BaderKit` to be easily inserted into complex workflows where QTAIM analysis is only one part of the process. For users who are less familiar with Python, `BaderKit` includes a command-line interface built with the `Typer` (Ramírez, 2019) package for quick one-off calculations, and a simple desktop application built with `PyQt5` (Computing, 2016) and `PyVista` (Sullivan & Kaszynski, 2019) for visualization.

A secondary goal of `BaderKit` is to update grid-based Bader algorithms to utilize modern architectures. To achieve this, `BaderKit` uses the `Numba` (Lam et al., 2015) and `NumPy` (Harris et al., 2020) packages to compile expensive operations to machine code and allow for fast, C-based, vectorized calculations. To improve speed further on modern multi-core CPUs, each Bader algorithm has been reworked from the ground up to allow for parallelization where possible.

Speed and Parallelization

BaderKit includes each of the algorithms from the original Fortran code including the `ongrid` (Henkelman et al., 2006), `neargrid` (Tang et al., 2009), and `weight` (Yu & Trinkle, 2011) methods. On a modern machine (AMD Ryzen™ Threadripper™ 1950X CPU with 16 cores), `BaderKit` runs two to three times as fast as the original Fortran code (Figure 1). This speedup is primarily due to changes made to each method that allow them to be parallelized on multi-core architectures. Here, we briefly touch on the most significant changes.

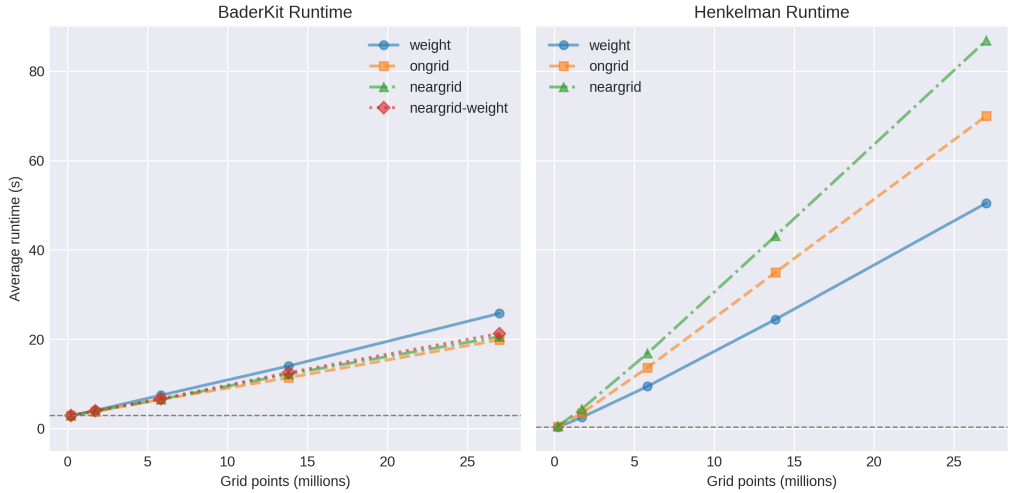

Figure 1: Comparison of runtimes for BaderKit and the Henkelman Fortran code calculated by taking the average of 10 runs. For a fair comparison, both methods were called through the command line and include reading files, running the algorithm, and writing outputs.

The original `ongrid` and `neargrid` methods perform hill climbing algorithms that start at arbitrary points and climb the steepest gradient until either a maximum or previous point is reached. The `ongrid` method is fast, but its results are highly dependent on the rotational

orientation of the system. The `neargrid` method is similar, but improves upon this by storing a correction vector from the current point to the true gradient and making adjustments when the vector is sufficiently large. It requires an additional edge refinement step as the correction vector is only correct for the initial starting point of the path. Instead of a path-building method, `BaderKit` loops over each point in parallel and establishes a pointer to the steepest neighbor. This creates a classic 'forest of trees' problem where the root of each tree corresponds to a Bader basin. `BaderKit` then finds these roots using a vectorized pointer jumper algorithm. For the `neargrid` method, the points along the edge are then iteratively refined in parallel using the original path method. This operation is significantly sped up by caching the gradients calculated during the initial pointer construction.

In contrast to the `ongrid` and `neargrid` methods, the `weight` method allows points to be assigned to multiple Bader basins. In the original algorithm, the points are sorted then looped over from high to low values. At each point, the algorithm calculates a flux representing the fraction of the point flowing to each of its neighbors, then uses the results from those neighbors to calculate the fraction of the point flowing to each basin. Though the loop from high to low must be done serially, `BaderKit` improves upon the original by calculating the fluxes in parallel. Because, this flux is only important at points that straddle multiple basins, `BaderKit` performs an initial fast loop that assigns interior points without calculating the flux. Additionally, we have developed a new hybrid method we call `neargrid-weight`. Since only the flux at basin edges is important, we first find interior points using the faster, fully parallelized `neargrid` method, then obtain fractional assignments at the edges using the `weight` method.

Basin Reduction

In addition to improved speed, BaderKit fixes an issue in the original codes handling of local maxima. In highly symmetrical systems, it is common for a local maximum to sit precisely between two or more grid points. This results in multiple adjacent points with values greater than or equal to their neighbors. In the original Fortran code, these points are incorrectly considered separate maxima. `BaderKit` explicitly checks for this, combines adjacent maxima, and performs a quick refinement using a parabolic fit to estimate the true, offgrid, location of the maxima (Figure 2). This adds negligible time, and results in more physically reasonable basins.

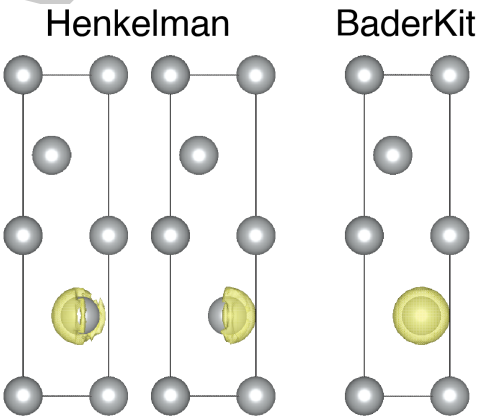

Figure 2: Comparison of basins found around an Ag atom in the Henkelman code and BaderKit. BaderKit merges the voxelated basins into a single maximum

Research Impact Statement

BaderKit was designed to provide researchers with easier access to underlying aspects of the Bader algorithm. It has already been incorporated into the BadELF(Weaver et al., 2023) code and is currently being expanded to include tools for detailed topological analysis of the electron localization function. The structure of BaderKit allows researchers with building blocks to construct further complex charge and topology analyses that would be difficult or impossible with other currently available software.

Though currently relatively limited in scope, BaderKit is set up to allow easy contribution from others. As new functionality is requested, this will allow BaderKit to expand to meet the needs of the chemistry and materials communities.

AI Usage Disclosure

No generative AI tools were used in the writing of this manuscript, preparation of supporting materials, or code documentation. OpenAI's GPT-5 model was occasionally used to assist in code development. A human developer made all core design decisions and reviewed, edited, and tested any generated code.

Acknowledgements

S.M.W. acknowledges support of this research by the NSF Graduate Research Fellowship Program. While developing this software, computational resources were provided, in part, by the Research Computing Center at the University of North Carolina at Chapel Hill.

References

Computing, R. (2016). *PyQt5*. https://www.riverbankcomputing.com/software/pyqt/

Frisch, M. J., Trucks, G. W., Schlegel, H. B., Scuseria, G. E., Robb, M. A., Cheeseman, J. R., Scalmani, G., Barone, V., Petersson, G. A., Nakatsuji, H., Li, X., Caricato, M., Marenich, A. V., Bloino, J., Janesko, B. G., Gomperts, R., Mennucci, B., Hratchian, H. P., Ortiz, J. V., … Fox, D. J. (2016). *Gaussian˜16 Revision C.01*. Gaussian Inc. Wallingford CT. gaussian.com

Giannozzi, P., Baroni, S., Bonini, N., Calandra, M., Car, R., Cavazzoni, C., Ceresoli, D., Chiarotti, G. L., Cococcioni, M., Dabo, I., & al., et. (2009). QUANTUM ESPRESSO: A modular and open-source software project for quantum simulations of materials. *Journal of Physics. Condensed Matter*, *21*(39), 395502. https://doi.org/10.1088/0953-8984/21/39/395502

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

Henkelman, G., Arnaldsson, A., & Jónsson, H. (2006). *A fast and robust algorithm for bader decomposition of charge density*. *36*, 354–360. https://doi.org/10.1016/j.commatsci.2005.04.010

Kerrigan, A. M. (2020). Pybader. In *GitHub repository*. GitHub. https://github.com/adam-kerrigan/pybader

Kresse, G., & Furthmüller, J. (1996). Efficient iterative schemes for ab initio total-energy calculations using a plane-wave basis set. *Physical Review B*, *54*(16), 11169–11186. https://doi.org/10.1103/PhysRevB.54.11169

Lam, S. K., Pitrou, A., & Seibert, S. (2015). Numba: A LLVM-based python JIT compiler. In *Proceedings of the second workshop on the LLVM compiler infrastructure in HPC - LLVM '15* (pp. 1–6). ACM Press. https://doi.org/10.1145/2833157.2833162

Ong, S. P., Richards, W. D., Jain, A., Hautier, G., Kocher, M., Cholia, S., Gunter, D., Chevrier, V. L., Persson, K. A., & Ceder, G. (2013). Python materials genomics (pymatgen): A robust, open-source python library for materials analysis. *Computational Materials Science*, *68*, 314–319. https://doi.org/10.1016/j.commatsci.2012.10.028

Otero-de-la-Roza, A., Johnson, E. R., & Luaña, V. (2014). Critic2: A program for real-space analysis of quantum chemical interactions in solids. *Computer Physics Communications*, *185*(3), 1007–1018. https://doi.org/10.1016/j.cpc.2013.10.026

Ramírez, S. (2019). Typer: Build great CLIs. Easy to code. Based on python type hints. In *GitHub repository*. GitHub. https://github.com/fastapi/typer

Sullivan, C., & Kaszynski, A. (2019). PyVista: 3D plotting and mesh analysis through a streamlined interface for the visualization toolkit (VTK). *The Journal of Open Source Software*, *4*(37), 1450. https://doi.org/10.21105/joss.01450

Tang, W., Sanville, E., & Henkelman, G. (2009). A grid-based bader analysis algorithm without lattice bias. *Journal of Physics. Condensed Matter*, *21*(8), 084204. https://doi.org/10.1088/0953-8984/21/8/084204

Weaver, S. M., Sundberg, J. D., Slamowitz, C. C., Radomsky, R. C., Lanetti, M. G., McRae, L. M., & Warren, S. C. (2023). Counting electrons in electrides. *Journal of the American Chemical Society*, *145*(48), 26472–26476. https://doi.org/10.1021/jacs.3c10876

Yu, M., & Trinkle, D. R. (2011). Accurate and efficient algorithm for bader charge integration. *The Journal of Chemical Physics*, *134*(6), 064111. https://doi.org/10.1063/1.3553716