Journal of Physics: Condensed Matter

PAPER • OPEN ACCESS

f90wrap: an automated tool for constructing deep Python interfaces to modern Fortran codes

To cite this article: James R Kermode 2020 *J. Phys.: Condens. Matter* **32** 305901

OPEN ACCESS
IOP Publishing
Journal of Physics: Condensed Matter
J. Phys.: Condens. Matter **32** (2020) 305901 (6pp)
https://doi.org/10.1088/1361-648X/ab82d2

f90wrap: an automated tool for constructing deep Python interfaces to modern Fortran codes

James R Kermode

Warwick Centre for Predictive Modelling, School of Engineering, University of Warwick, Coventry CV4 7AL, United Kingdom

E-mail: J.R.Kermode@warwick.ac.uk

Received 3 January 2020, revised 9 March 2020
Accepted for publication 24 March 2020
Published 5 May 2020

Abstract

`f90wrap` is a tool to automatically generate Python extension modules which interface to Fortran libraries that makes use of derived types. It builds on the capabilities of the popular `f2py` utility by generating a simpler Fortran 90 interface to the original Fortran code which is then suitable for wrapping with `f2py`, together with a higher-level Pythonic wrapper that makes the existance of an additional layer transparent to the final user. `f90wrap` has been used to wrap a number of large software packages of relevance to the condensed matter physics community, including the `QUIP` molecular dynamics code and the `CASTEP` density functional theory code.

Keywords: Fortran, Python, f2py, interoperability, interfacing, wrapping codes

(Some figures may appear in colour only in the online journal)

1. Introduction

Modern scientific computing relies on the existence of many well-documented software libraries. This has led to a new programing paradigm based around interfacing of existing libraries and packages, which allows existing tools to be combined to produce something that is more than the sum of the constituent parts [1]. Python has emerged as the *de facto* standard 'glue' language [2] for these purposes, meaning that any code that has a Python interface can be combined with others in complex and imaginative ways which often go beyond the original intentions of the developers. The success of Python in this domain is largely thanks to the well-developed ecosystem of scientific packages (e.g. `numpy` [3], `scipy` [4], `matplotlib` [5], the `Jupyter` [6] framework which encourages literate programing and reproducible research, and the `anaconda` distribution and package management system).

Despite the rise in popularity of high-level programing languages, the majority of high performance computing codes in use across the computational modelling community today—and within the domain of computational condensed matter physics in particular—are monolithic programs written in low-level languages such as C or Fortran, focused on performing a single task specified by the users input file. In standard usage patterns, outputs from codes run at HPC centres are typically stored in human-readable format to a text file or, increasingly, in structured formats such as XML, JSON and HDF5 before transferring to a workstation for subsequent analysis. While this traditional mode of operation has served the scientific community well for decades, it has become clear that by adopting a more modern approach, in which the codes serve as software engines tied into a larger and heterogeneous production environment, both the ease and the rate of gaining new functionality would be much higher, meaning that the real world impact of simulation codes could be significantly enhanced.

While the task of Fortran-to-Python interface generation addressed in this article is general to a wide range of computational modelling domains, the examples presented fall within the condensed matter physics/materials modelling domain. This is of particularly importance given the increasing requirement to interface codes to one another to underpin and support the recent upsurge in usage of data-driven and machine-learning approaches across computational modelling. The challenges and solution methods discussed here are of course still applicable to a much wider domain of application.

1.1. File-based interfaces

A number of successful projects enable interoperability through file-based communication between codes, with scripts generating input files and parsing output from off-the-shelf versions of packages.

Prominent examples of interoperable packages within the field of condensed matter include the Atomic Simulation Environment (ASE) [7] and Py-ChemShell [8]. ASE is regarded as the state-of-the-art in this field, as it provides the widest range of 'calculators', enabling many electronic structure and force field codes to be used as drop-in replacements. High-level functionality can be coded generically or imported from other packages such as `spglib` [9] for space-group symmetry or `phonopy` [10] for phonon calculations. However, the reliance on file-based interfaces requires substantial time investment to maintain a library of input generators and output parsers. Having such a collection of parsers aids validation and verification supporting activities such as the Δ-code project, where a cross-comparison of density functional theory (DFT) codes was carried out [11], as well as providing opportunities to normalize simulation data to a common format suitable for sharing and reuse [12].

Interoperability is a key requirement to support a variety of aims including: (i) improved software sustainability; (ii) verification and validation [11]; (iii) multiscale materials modelling where several codes are coupled [13]; (iv) machine learning [14]; and (v) uncertainty quantification [15].

1.2. Deep scripting interfaces

For efficient coupling of codes, or to enable access to all features of a complex code, a direct programmatic interface is needed, going beyond the capabilities of the file-based interfaces discussed above. Moreover, file-based interfaces can be slow and/or incomplete and robust parsers have proven hard to write and costly to maintain. Standardised output (e.g. chemical markup language [16]) and next-generation parsers are part of the solution: for example the NOMAD centre of excellence has produced parsers for many widely used electronic structure codes [12].

Modern scientific codes that follow good software engineering principles consist of self-contained modules that implement a well-defined and slowly evolving API (application programming interface). The top-level programme is (conceptually) a script that connects user input files to this API. This suggests an alternative approach to interoperability, namely using the existing modular structure to provide a much deeper wrapping, exposing the full public API of each code to a wide community of script writers who do not need familiarity with the underlying code. In this way, efficiency can be maintained, e.g. for DFT codes, keeping the previous solution in memory enables efficient wavefunction extrapolation from one timestep to the next.

In general, the deep interface approach brings further, much wider benefits. These include the expansion of access to advanced features to less experienced programmers, the simplification of top-level programs using efficient algorithms available in high-level languages, enhanced introspection and visualisation capabilities, and the immediate provision of a full unit and regression testing framework. The latter enables continuous integration, resulting in higher quality and more sustainable code bases. Further benefits include the encouragement of good software engineering practices—modularity, re-entrancy and stable APIs—and speeding up the development of new algorithms using an optimal mix of high- and low-level languages. High-level scripting also promotes open data efforts, because the barriers (both technical and legal) to publishing scripts that lead to new scientific results are much lower than within the monolithic software package framework.

Despite recent increases in the use of high-level languages, the scientific computing community is fortunate to possess a great deal of high-quality and well-maintained Fortran code. For example, analysis of usage data from the UK national supercomputer centre, ARCHER, collected by EPCC shows that approximately 70% of recent CPU usage has come from Fortran codes, compared to around 7% for C++ codes and around 6% for C codes [17].

Adding deep Python interfaces to existing codes makes optimal use of this resource, increasing the sustainability of the software infrastructure. Future proofing goes further, as thanks to the dominance of Python anything accessible from Python will be available to newly emerging high-level technical languages such as Julia [18]. While there are many automatic interface generators for C++ codes (e.g. `SWIG` [19] or `Boost.Python` [20]), support for modern Fortran is much more limited despite its widespread use in scientific computing. The `f2py` [21] interface generator supports many aspects of Fortran 77/90/95 codes. This allows individual routines or simple libraries to be wrapped in a portable, compiler independent way with good array support. However, `f2py` has no support for modern Fortran features such as derived types (cf classes in C++) or overloaded interfaces.

The open source `f90wrap` package described in this article adds derived type support to `f2py` using an additional layer of wrappers. `f90wrap` also exposes module data, provides efficient array access with no copying of data, and supports Python 2.7+ and 3.x. A schematic overview of the process of wrapping a modular Fortran code is shown in figure 1. The approach builds on techniques developed in the `QUIP` (Quantum mechanics and interatomic potentials) code [22], which originated as a pure Fortran 95 library but has had a full deep Python interface–`quippy`—since 2009 that gives full access to public subroutines, derived types and data. Recently

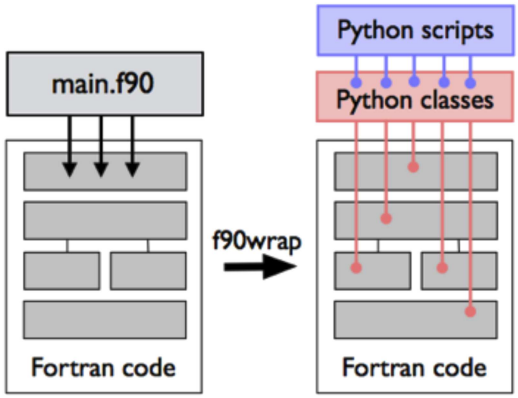

Figure 1. Schematic overview of the usage of f90wrap to expose modular Fortran codes to Python.

more and more high-level functionality has been coded in Python, and/or moved over to the widely-used ASE framework discussed above.

The rest of the article is organised as follows: section 2 provides an overview of the method of solution employed; section 3 give details of the usage and features of the package; section 4 presents case studies demonstrating the applicability of `f90wrap`.

2. Methodology

There are five key step in the process of wrapping a Fortran package with `f90wrap` to allow it to be called from Python:

(a) The Fortran source files are scanned, building up an abstract symbol tree (AST) which describes all the modules, types, subroutines, functions and interfaces found.

(b) The AST is transformed to remove nodes which should not be wrapped (e.g. private symbols in modules, routines with arguments of a derived type external to the project, etc).

(c) The `f90wrap.f90wrapgen.F90WrapperGenerator` class is used to write a simplified Fortran 90 prototype for each routine, with derived type arguments replaced by integer arrays containing a representation of a pointer to the derived type, in the manner described in [23]. This allows opaque references to the underlying Fortran derived type data structures to be passed back and forth between Python and Fortran.

(d) `f2py` is used to combine the F90 wrappers and the original compiled functions into a Python extension module (optionally, f2py can be replaced by `f2py-f90wrap`, a slightly modified version which adds support for exception handling and interruption during execution of Fortran code, as described in more detail below).

(e) The `f90wrap.pywrapgen.PythonWrapperGenerator` class is used to write a thin object-oriented layer on top of the `f2py`-generated wrapper functions which handles conversion between Python object instances and Fortran derived-type variables, converting arguments back and forth automatically.

3. Usage and features of f90wrap

3.1. Installation and basic usage

`f90wrap` is a registered Python package and is available from the Python package index. Source code is available under version 3 of the Lesser General Public Licence (LGPL) from GitHub [24]. It can be used with either Python 2.7 or 3.x and installed via the command

3.1.1. `pip install f90wrap`. To use `f90wrap` to wrap a set of Fortran 90 source files and produce wrappers suitable for input to f2py use:

3.1.2. `f90wrap -m MODULE F90_FILES`. where `MODULE` is the name of the Python module to be produced (e.g. the name of the Fortran code you are wrapping) and `F90_FILES` is a list of Fortran source files containing the modules, types and subroutines the user would like to expose via Python.

This will produce two types of output: Fortran 90 wrapper files suitable for input to `f2py` to produce a low-level Python extension module (typically named `_MODULE.so`), and a high-level Python package named `MODULE` designed to be used together with the `f2py`-generated module to give a more Pythonic interface.

One Fortran 90 wrapper file is written for each source file, named `f90wrap_F90_FILE.f90`, plus possibly an extra file named `f90wrap_toplevel.f90` if there are any subroutines or functions defined outside of modules in `F90_FILES`.

To use `f2py` to compile these wrappers into an extension module, use:

3.1.3. `f2py -c -m _MODULE OBJ_FILES f90wrap_*.f90 *.o LINK_OPTIONS`. where `_MODULE` is the name of the low-level extension module to be produced, and `LINK_OPTIONS` can be used to link in other dependent libraries which are required at runtime but do not to be explicitly exposed to Python (e.g. mathematics libraries such as BLAS and LAPACK).

Optionally, one can replace `f2py` with `f2py-f90wrap`, which is a slightly modified version of `f2py` distributed with `f90wrap` that introduces some additional features:

(a) Allows the Fortran `present()` intrinsic function to work correctly with optional arguments. If an argument to an f2py wrapped function is optional and is not given, it is replaced with a `NULL` pointer.

(b) Allows Fortran routines to raise a Python `runtimeerror` exception with a message by calling an external function `f90wrap_abort()`. This is implemented using a `setjmp()/longjmp()` trap. This allows internal errors to be caught and translated into Python exceptions.

(c) Allows Fortran routines to be interrupted with `ctrl+C` by installing a custom interrupt handler before the call

into Fortran is made. After the Fortran routine returns, the previous interrupt handler is restored.

3.2. Additional features

Documentation is automatically extracted from the source code and made available to the user through Python documentation strings ('docstrings'), enhancing the productivity of the development environment. This can be rendered into user documentation using the `Sphinx` package.

A variety of additional command line arguments can be passed to `f90wrap` to customize how the wrappers are generated; these are described in the online documentation.

3.3. Limitations

There are a number of limitations of the wrapping approach, some of which are fundamental and others which could be addressed by adding additional functionality in future.

(a) Unlike standard `f2py`, `f90wrap` converts all `intent(out)` array arguments to `intent(in, out)`. This is a deliberate design decision to allow allocatable and automatic arrays of unknown output size to be used from Python. It is impossible in general to infer what size array needs to be allocated, so relying on the user to pre-allocate arrays from Python is the safest solution.

(b) Scalar arguments without a specified intent are treated as `intent(in)` by `f2py`. To have `inout` scalars, one can call `f90wrap` with the `-default-to-inout` flag and declare the Python variables as single-element numpy arrays (using `numpy.zeros(1)`, for example).

(c) Pointer arguments are not currently supported. This is not a fundamental limitation and could be addressed in future.

(d) Arrays of derived types are not yet fully supported: a workaround is provided for 1D-fixed-length arrays, i.e. `type(a), dimension(b) :: c`. This is also not a fundamental limitation.

4. Case studies

`f90wrap` is capable of wrapping large, complex codes and providing deep access to all internal data. The examples presented below are taken from materials modelling, but the applicability of `f90wrap` is general. It is now used extensively in the `QUIP` [22] and `CASTEP` codes [25].

4.1. Wrapping the Bader code

As an example of what can be achieved, an f90wrap wrapper for the `Bader` charge analysis code [26, 27] has been produced. This is a widely used code for post-processing charge densities to construct Bader volumes and compute the total charge associated with each volume. The addition of a Python interface allows the code to be used in workflows without the burdens of input/output and file-format conversion.

To produce this package, it was sufficient to download the source code [28] and use `f90wrap` to automatically generate a deep Python interface with very little manual work, using the commands below:

```
f90wrap -v -k kind_map -I init.py -m bader\
kind_mod.f90 matrix_mod.f90 \
ions_mod.f90 options_mod.f90 charge_mod.f90 \
chgcar_mod.f90 cube_mod.f90 io_mod.f90 \
bader_mod.f90 voronoi_mod.f90 multipole_mod.f90
f2py-f90wrap -c -m _bader f90wrap_*.f90 -L. -lbader
```

The first line generates Python interfaces to the listed Fortran 90 modules, with the additional of a small amount of hand-written Python code (`init.py`) and a mapping from Fortran to Python types provided in the file `kind_map`. The name of the output module is specified with the `-m` argument as `Bader`.

As a brief demonstration of the utility of the new wrapper, we show how to restart a `GPAW` [29] DFT calculation, here for an 8 atom silicon crystal, and retrieve the density, noting that `GPAW` does not intrinsically have the ability to compute Bader charges.

```
import bader
from gpaw import restart

# read in charge density
si, gpaw = restart('si-vac.gpw')
rho = gpaw.get_pseudo_density()

# use wrapped Bader code to post-process
bdr = bader.bader(si, rho)

# collect charge density associated with atom #3
atom = 3
rho3 = np.zeros_like(rho)
for v in (bdr.nnion == atom+1).nonzero()[0]:
rho3[bdr.volnum == v+1] = rho[bdr.volnum == v+1]
```

This script uses the Python interface to construct a Bader partitioning of the charge density, and then accesses members of the `bdr` Fortran derived-type instance (e.g. `bdr.nnion`, the indices of ions associated with particular volumes, and `bdr.volnum`, the number associated with each volume) to extend the capabilities of the original code, here in a trivial way to collect together the parts of the charge density associated with a particular atom as illustrated in figure 2, but in principle this could be much more complex. The modified version of the `Bader` code including the automatically generated Python interface is available from [30].

4.2. Wrapping the QUIP and GAP codes

`f90wrap` is used to generate the `quippy` wrappers to the quantum mechanics and interatomic potentials (QUIP) and Gaussian approximation potential (GAP) codes [31] which are used for advanced molecular simulations such as training and using machine learning interatomic potentials [14]. `f90wrap` actually grew out of the need to expose these Fortran codes to Python to allow interoperability with other tools such as the Atomic Simulation Environment (ASE); originally the same

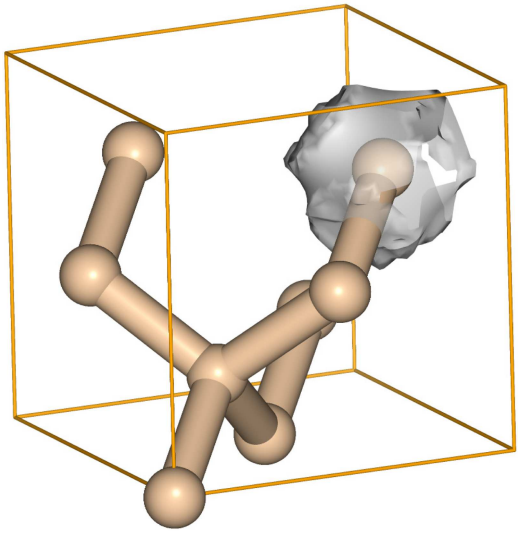

Figure 2. Illustration of the application of `f90wrap` to compute Bader partitioning of the charge density from a DFT calculation of silicon with the `GPAW` code, which does not natively provide this functionality. The charge density associated with one atom is shown with an isosurface.

functionality was implemented directly within the QUIP code, but more recently QUIP has been switched over to call the rewritten `f90wrap` directly. This brings advantages in terms of modularity and, crucially, Python 3 compatibility. A number of examples and tutorials using QUIP and GAP from Python are given on the libatoms website [32].

4.3. Wrapping the CASTEP code

Starting in 2014 a Python interface to the CASTEP DFT code [33] was produced using `f90wrap`. The `CasPyTep` package provides functionality far in excess of what could be done manually—around 35 kLOC (thousand lines of code) Fortran and 55 kLOC Python are auto-generated, giving access to 20+ derived types and ∼2600 subroutines. Module-level variables such as the current simulation cell are exposed, with Fortran derived types visible as Python classes.

The interface allows atoms to be moved and calculations to be continued without either restarting from scratch or incurring the large I/O burden associated with checkpointing the full state of the code. This makes it quick to try out new high-level algorithms: for example, a general-purpose preconditioner for geometry optimisation [34] was implemented as a high-level Python code by making it possible to use it with CASTEP and other codes. A number of similar projects which make use of the Python interface to enable rapid prototyping are ongoing.

5. Discussion and conclusion

This article has motivated the case for exposing deep interface to Fortran codes to Python and analysed the benefits, as well as reviewing the `f90wrap` package which provides a practical tool to realise this ambition. It is already capable of wrapping large Fortran codes, as demonstrated by the case studies discussed above.

From a pedagodic perspective, one might question whether providing Python interface layers will take users further away from the Fortran source code. This objection presupposes users look at source code; typical industrial users often do not even have access to the source code, and typically academic users simply compile and run it. In contrast, Python interfaces allow everyone to prototype developments rapidly. Of course, HPC and low-level developers still need to code in Fortran, but by opening up scientific tools to a broader community, it would become feasible for new users such as experimentalists to develop impressive, high-level code without having to understand Fortran at all. In this way, `f90wrap` provides an enabling technology, opening up software development to new developers.

While the principal users of `f90wrap` are currently intended to be code developers, in the longer term this could become a wider usage pattern suitable for all users, since the computational overhead of wrapping Fortran code and exposing it to Python is relatively light. For interactive use, serial execution is convenient, but for complex tasks parallelization of the Fortran code is typically required. This can still be done though the wrapper, using either OpenMP (multiple threads) or MPI (multiple processes). In the former case, usage is straightforward; the Fortran code must simply by compiled with OpenMP support enabled. While Python's global interpreter lock (GIL) prevents multi-threading within Python code, multithreading is still possible in OpenMP regions inside Fortran function calls; care must be taken however, to ensure that sufficient time is spent within each call to offset the overhead of starting and pausing threads. Wrapping MPI Fortan code is a little more complex, since the Python interpreter must be configured to act as the root MPI task, but this is eminentally achievable.

In recent years many fundamental science software tools have come to maturity. Scripting interfaces are already very useful for automating calculations, but taking this further to connect components in new ways would help to give legacy Fortran code a new lease of life. It is hoped that the capabilities of `f90wrap` will allow it to be part of this exciting development.

Acknowledgments

I would like to thank the many contributors and users of the `f90wrap` GitHub repository, as well as support and encouragement from mentors and early users: Sandro De Vita, Gábor Csányi and Noam Bernstein. In particular, Tamas Stenzcel has made significant recent contributions to `f90wrap` while working on the `QUIP` use case. The `CasPyTep` application has benefited from support and encouragement from the CASTEP Developers Group, in particular Phil Hasnip and Matt Probert. Greg Corbett carried out the initial work on this use case, and Sebastian Potthoff added MPI support to `CasPyTep` and optimised the performance of the nudged elastic band algorithm. I acknowledge useful discussions with members of the UK Car Parrinello

Consortium, in particular David Bowler and Chris Skylaris in addition to those listed above. This work was in part supported by the EPSRC under grants EP/P002188/1, EP/L014742/1, EP/J022055/1 and EP/R043612/1.

References

[1] Ousterhout J K 1998 *Computer* **31** 23–30
[2] Oliphant T E 2007 *Comput. Sci. Eng.* **9** 10–20
[3] Oliphant T E 2006 *A Guide to NumPy* (USA: Trelgol Publishing)
[4] Jones E *et al* 2001 *SciPy: Open Source Scientific Tools for Python* URL: http://scipy.org (accessed 30 July 19)
[5] Hunter J D 2007 *Comput. Sci. Eng.* **9** 90–5
[6] Kluyver T *et al* 2016 Jupyter notebooks – a publishing format for reproducible computational workflows *Positioning and Power in Academic Publishing: Players, Agents and Agendas* ed F Loizides and B Schmidt (IOS Press) pp 87–90
[7] Larsen A H *et al* 2017 *J. Phys.: Condens. Matter* **29** 273002
[8] Lu Y, Farrow M R, Fayon P, Logsdail A J, Sokol A A, Catlow C R A, Sherwood P and Keal T W 2019 *J. Chem. Theor. Comput.* **15** 1317–28
[9] Togo A and Tanaka I 2018 arXiv:1808.01590
[10] Togo A and Tanaka I 2015 *Scripta Mater* **108** 1–5
[11] Lejaeghere K *et al* 2016 *Science* **351** aad3000
[12] Ghiringhelli L M, Carbogno C, Levchenko S, Mohamed F, Huhs G, Lüders M, Oliveira M and Scheffler M 2017 *npj Comput. Mater.* **3** 46
[13] Bernstein N, Kermode J R and Csányi G 2009 *Rep. Prog. Phys.* **72** 026501
[14] Bartók A P, Payne M C, Kondor R and Csányi G 2010 *Phys. Rev. Lett.* **104** 136403
[15] Aldegunde M, Kermode J R and Zabaras N 2016 *J. Comput. Phys.* **311** 173–95
[16] Murray-Rust P and Rzepa H S 2011 *J. Cheminf.* **3** 44
[17] ARCHER team 2019 *Archer Application Usage* URL: http://archer.ac.uk/status/codes (accessed 30 July 19)
[18] Bezanson J, Edelman A, Karpinski S and Shah V B 2017 *SIAM Rev.* **59** 65–98
[19] Beazley D M 2003 *Future Generat. Comput. Syst.* **19** 599–609
[20] Abrahams D and Grosse-Kunstleve R W 2003 Building hybrid systems with Boost.Python *C Users J. Arch.* **21** 29–36
[21] Peterson P 2009 *Int. J. Comput. Sci. Eng.* **4** 296–305
[22] Csányi G, Winfield S, Kermode J R, De Vita A, Comisso A, Bernstein N and Payne M C 2007 *IoP Comput. Phys. Newsletter* Spring 2007
[23] Pletzer A, McCune D, Maszala S, Vadlamani S and Kruger S 2008 *Comput. Sci. Eng.* **10** 86
[24] Kermode J 2020 *Source Code for f90wrap* URL: https://github.com/jameskermode/f90wrap
[25] Clark S J, Segall M D, Pickard C J, Hasnip P J, Probert M I J, Refson K and Payne M C 2005 *Z. Kristallogr.* **220** 567–70
[26] Bader R F W 1985 *Acc. Chem. Res.* **18** 9–15
[27] Tang W, Sanville E and Henkelman G 2009 *J. Phys.: Condens. Matter.* **21** 084204
[28] Henkelman G 2020 *Original Version of Bader Charge Analysis Code* URL: http://theory.cm.utexas.edu/henkelman/code/bader
[29] Mortensen J J, Hansen L B and Jacobsen K W 2005 *Phys. Rev.* B **71** 035109
[30] Kermode J 2020 *Modified Version of Bader Charge Analysis Code with Python Interface* URL: http://gitlab.com/jameskermode/bader
[31] libAtoms/QUIP collaboration 2006–2020 *Source Code for Quip* URL: http://github.com/libAtoms/QUIP
[32] Kermode J R, Bartok Partay A, Bernstein N and Csanyi G 2006–2020 *Libatoms, Quip and Gap* URL: http://libatoms.github.io
[33] Corbett G, Kermode J, Jochym D and Refson K 2015 *Rutherford Appleton Laboratory Technical Reports* URL: https://epubs.stfc.ac.uk/work/18048381
[34] Packwood D, Kermode J, Mones L, Bernstein N, Woolley J, Gould N, Ortner C and Csányi G 2016 *J. Chem. Phys.* **144** 164109