Atomic Simulation Recipes

A Python framework and library for automated workflows

Gjerding, Morten; Skovhus, Thorbjørn; Rasmussen, Asbjørn; Bertoldo, Fabian; Larsen, Ask Hjorth; Mortensen, Jens Jørgen; Thygesen, Kristian Sommer

Published in:
Computational Materials Science

Link to article, DOI:
10.1016/j.commatsci.2021.110731

Publication date:
2021

Document Version
Publisher's PDF, also known as Version of record

Computational Materials Science 199 (2021) 110731

Contents lists available at ScienceDirect

Computational Materials Science

journal homepage: www.elsevier.com/locate/commatsci

Atomic Simulation Recipes – A Python framework and library for automated workflows

Morten Gjerding *, Thorbjørn Skovhus, Asbjørn Rasmussen, Fabian Bertoldo, Ask Hjorth Larsen, Jens Jørgen Mortensen, Kristian Sommer Thygesen

Computational Atomic-scale Materials Design (CAMD), Department of Physics, Technical University of Denmark, 2800 Kgs. Lyngby, Denmark

ARTICLE INFO

Keywords:
High-throughput
Database
Data provenance
Workflow
Python
Materials computation
Density functional theory

ABSTRACT

The Atomic Simulation Recipes (ASR) is an open source Python framework for working with atomistic materials simulations in an efficient and sustainable way that is ideally suited for high-throughput projects. Central to ASR is the concept of a Recipe: a high-level Python script that performs a well defined simulation task robustly and accurately while keeping track of the data provenance. The ASR leverages the functionality of the Atomic Simulation Environment (ASE) to interface with external simulation codes and attain a high abstraction level. We provide a library of Recipes for common simulation tasks employing density functional theory and many-body perturbation schemes. These Recipes utilize the GPAW electronic structure code, but may be adapted to other simulation codes with an ASE interface. Being independent objects with automatic data provenance control, Recipes can be freely combined through Python scripting giving maximal freedom for users to build advanced workflows. ASR also implements a command line interface that can be used to run Recipes and inspect results. The ASR Migration module helps users maintain their data while the Database and App modules makes it possible to create local databases and present them as customized web pages.

1. Introduction

As computing power continues to increase and the era of exascale approaches, the development of software solutions capable of exploiting the immense computational resources becomes a key challenge for the scientific community. In the field of materials science, ab initio electronic structure (aiES) calculations are increasingly being conducted in a high-throughput fashion to screen thousands of materials for various applications [1–16] and to generate large reference data sets for training machine learning algorithms to predict fundamental materials properties [17–22] or design interatomic potentials [23–26]. The results from such aiES high-throughput calculations are often stored in open databases allowing the data to be efficiently shared and deployed beyond the original purpose [27–37].

While a few thousands of calculations can be managed manually, a paradigm in which data drives scientific discovery calls for dedicated workflow solutions that automatically submit and retrieve the calculations, store the results in organized data structures, and keep track of the origin, history and dependencies of all data, i.e. the data provenance. Ideally, the workflow should also attach explanatory descriptions to the data that allows them to be easily accessed, understood, and deployed – also by users with limited domain knowledge.

Materials scientists from the aiES community are employing a large and heterogeneous set of simulation codes based mainly on density functional theory (DFT) [38]. These codes differ substantially in the way they implement and solve the fundamental physical equations. This is due to the fact that different types of problems require different numerical approaches, e.g. high accuracy vs. large system sizes, periodic vs. finite vs. open boundary conditions, or ground state vs. excited state properties. In principle, the large pool of available aiES codes provides users with a great deal of flexibility and freedom to pick the code that best suits the problem at hand. In practice, however, the varying numerical implementations and the diverse and often rudimentary user interfaces make it challenging for users to switch between the different aiES codes leading to a significant “code barrier”.

To some extent, a similar situation exists with respect to materials properties. Although aiES codes provide access to a rich variety of physical and chemical properties, individual researchers often focus on properties within a specific scientific domain. While this may be sufficient in many cases, several important contemporary problems

* Corresponding author.
E-mail address: mortengjerding@gmail.com (M. Gjerding).

https://doi.org/10.1016/j.commatsci.2021.110731
Received 29 April 2021; Received in revised form 14 July 2021; Accepted 15 July 2021
Available online 5 August 2021

addressed by the aiES community are multi-physical in nature and require properties and insights from several domains. For example, evaluating the potential of a material as a photocatalyst involves an assessment of solar light absorption, charge transport, and chemical reactions at a solid–liquid interface. Calculating new types of properties for the first time is often a time-consuming process involving trial and error and the acquisition of technical, implementation-specific knowledge of no direct benefit for the user or the overall project aim. This situation may result in a "property barrier" that hampers researchers' exploitation of the full capacity of aiES codes.

In this paper, we introduce The Atomic Simulation Recipes (ASR) – a highly flexible Python framework for developing and working with computational materials workflows. The ASR reduces code and property barriers and makes it easy to perform high-throughput computations with advanced workflows while adhering to the FAIR Data Principles [39]. There are already some workflow solutions available in the field, some of the most prominent being AFlow [30], Fireworks [40], AiiDA [41], and Atomate [42]. However, these are either designed for one specific simulation code and/or constitute rather colossal integrated entities, the complexity of which could represent an entry barrier to some users. The ASR differs from the existing solutions in several important ways, and we expect it to appeal to a large crowd of computational researchers, e.g. those with Python experience who like to develop their own personalized (workflow) scripts and databases, less experienced users who prefer plug-and-play solutions, and those who wish to apply non-standard methodologies, e.g. compute GW band structures or Raman spectra, but feel they lack the expertise required for using standard low-level codes.

The basic philosophy of ASR is to prioritize usability and simplicity over system perfection. More specifically, ASR is characterized by the following qualities:

- **Flexibility:** The Python scripting interface and high degree of modularity provide users with almost unlimited freedom for developing and deploying workflows.
- **Modularity:** The key components of ASR, namely the workflow development framework (ASR core), the Database and App modules, the task scheduler (MyQueue), and the simulation codes, are separate independent entities. Moreover, the Recipe library concept supports modular workflow designs and reuse of code.
- **Data locality:** Generated data is stored in a special folder named `.asr` where it can be accessed transparently via command line tools (similar to Git).
- **Compatibility:** For compatibility with external simulation codes, the ASR core is fully simulation code-independent while specific Recipe implementations communicate with simulation codes via the abstract ASE Calculator interface.
- **Minimalism and pragmatism:** ASR is based on simple solutions that work efficiently in practice. This makes ASR fast to learn, easy to use, and relatively uncomplicated to adapt to future demands.

At the core of ASR is the concept of a *Recipe*. In essence, a Recipe is a piece of code that can perform a certain simulation task (e.g. relax an atomic structure, calculate a Raman spectrum, or identify covalently bonded components of a material) while recording all relevant results and metadata. The use of Recipes makes it simple to run simulations from either Python or the command line. For example,

```
$ asr run "asr.bandstructure -atoms structure.json"
```

will calculate the electronic band structure of the material `structure.json`. Subsequently, the command

```
$ asr results asr.bandstructure
```

will produce a plot of the band structure. With two additional commands, the ASR results can be inspected in a web browser, see example in Fig. 6.

In practice, Recipes are implemented as Python modules building on the Atomic Simulation Environment (ASE) [43]. Recipes conform to certain naming and structured programming conventions, making them largely self-documenting and easy to read. To keep track of data provenance, Recipes utilize a caching mechanism that automatically logs all exchange of data with the user and other Recipes in a uniquely identifiable `Record` object. Not only does this guarantee the documentation and reproducibility of the results, it also allows ASR to determine whether a given Recipe task has already been performed (such that its result can be directly loaded and returned) and to detect if a Recipe task needs to be rerun because another piece of data in its dependency chain has changed. In addition, Recipes implement presentation and explanatory descriptions of their outputs and may also define a web panel for online presentation.

The Recipes of the current ASR library cover a variety of computational tasks and properties (see Table 1). Most of the 40+ available Recipes utilize DFT. However, some Recipes do not involve calls to a simulation code (e.g. symmetry analysis or construction of phase diagrams) while others employ beyond-DFT methodology (e.g. the GW method or the Bethe–Salpeter equation). These library Recipes can be used "out of the box" or modified to fit the user's need. New Recipes may be developed straightforwardly following the documentation and large body of available examples. Recipes can be combined into complex workflows using Python scripting for maximal flexibility and compatibility with ASE and other relevant Python libraries like PymatGen [44], Spglib [45] and Phonopy [46]. The Python workflows may be executed on supercomputers using the MyQueue [47] task scheduler front-end or other similar systems.

The ASR contains a number of tools for working with the ASE database module, which makes it easy to generate and maintain local materials databases. Relying on the Recipes' web panel implementations, these databases may be straightforwardly presented in a browser allowing for easy inspection, querying, and sharing of results on a local or public network. As an example of an ASR-driven database project we refer to the Computational 2D Materials Database (C2DB) [32,48].[1]

While the core of ASR, i.e. the Recipe concept and caching system, is fully simulation code-independent, most Recipe implementations of the current library contain calls to the specific aiES code GPAW [49]. At the moment, among the Recipes calling an external simulation code, only the asr.relax and asr.stiffness Recipes generalize to all ASE calculators, which support the calculation of stresses and atomic forces. We are currently working on a generalization of the ASE Calculator interface which will decouple Recipe implementations from simulation codes. In the future, many Recipes will therefore work with multiple simulation codes.

Another on-going effort is to generalize the organization of calculated results. For example results are currently presented mainly by material. This is practical for a database which primarily associates a number of properties with each material, but not for presenting sets of results parametrized over other variables than the material. These limitations will be removed over the next releases.

The rest of this paper is organised as follows: In Section 2 we provide a general overview of the main components of ASR. Section 3 zooms in on the central Recipe concept and its caching system while Section 4 gives an overview of the currently available Recipes. In Section 5, the Database and App modules are described. Section 6 gives a brief presentation of the Computational 2D Materials Database as an example of an ASR-driven high-throughput database project and provides a few concrete examples of Recipe implementations. Section 7 describes the different user interfaces supported by ASR while Sections 8 and 9 explain how ASR manages data migration and provenance, respectively. Sections 10 and 11 cover documentation and technical specifications.

[1] http://c2db.fysik.dtu.dk.

Table 1
List of Recipes currently implemented in the ASR library. Most of the Recipes depend explicitly on the GPAW electronic structure code. The Recipes are grouped under thematic headings and listed in alphabetic order.

Recipe name	Description
Atomic structure	
asr.database.duplicates	Remove duplicate structures from a database
asr.database.rmsd	Root mean square distance between structures
asr.dimensionality	Dimensionality of covalently bonded substructures of a material
asr.push	Push atoms along specific phonon mode
asr.relax	Relax atomic structure
asr.setup.defects	Generate native point defects
asr.setup.displacements	Generate structures with a single displaced atom
asr.setup.magnetize	Initialize atomic magnetic moments
asr.setup.reduce	Reduce supercell to primitive cell
asr.setup.symmetrize	Symmetrize an atomic structure
asr.structureinfo	Extract structural information
Thermodynamic properties	
asr.chc	Constrained convex hull stability analysis
asr.convex_hull	Convex hull stability analysis
asr.defectformation	Formation energy of neutral point defect
asr.fere	Fitted elemental reference energies
Mechanical properties	
asr.phonopy	Phonon band structure and dynamical stability
asr.piezoelectrictensor	Piezoelectric tensor
asr.stiffness	Stiffness tensor
Electronic properties	
asr.bader	Bader charge analysis
asr.bandstructure	Kohn–Sham band structure
asr.berry	Various band topology invariants
asr.borncharges	Born effective charge tensor
asr.deformationpotentials	Deformation potentials (only for 2D)
asr.dos	Density of states
asr.emasses	Effective masses
asr.fermisurface	Fermi surface
asr.formalpolarization	Formal polarization phase
asr.gs	Electronic ground state
asr.gw	G_0W_0 quasiparticle band structure
asr.hse	HSE06 band structure
asr.pdos	Orbital projected density of states
asr.proj_bandstructure	Orbital projected Kohn–Sham band structure
Magnetic properties	
asr.exchange	Magnetic exchange coupling
asr.magnetic_anisotropy	Magnetic anisotropy
asr.magstate	Determine magnetic state
Optical properties	
asr.bse	Optical absorption from Bethe–Salpeter Equation (BSE)
asr.infraredpolarizability	Infrared polarizability (caused by vibrations)
asr.plasmafrequency	Plasma frequency (from intraband transitions)
asr.polarizability	Optical polarizability (caused by electrons)
asr.raman	Raman spectrum (first-order)
asr.shg	Second harmonics generation
asr.shift	Shift current

Finally, Section 12 summarises the paper and presents our future perspectives for ASR.

2. Overview of ASR

Fig. 1 shows a schematic overview of the main components of the ASR and their mutual dependencies. An arrow from X to Y indicates a direct dependence of Y on X, e.g. via function calls (Y calls X). The ASR modules have been divided into the ASR core modules (Cache and Recipe) and the ASR user interfaces (command-line interface, Python, Task scheduler front-end, and Apps). In addition, the ASR Database and Data migration modules contain tools for working with databases and maintaining data, respectively.

Recipes implement specific, well defined materials simulation tasks as Python modules building on the ASE [43] and other Python libraries. A Recipe integrates with a Cache module that keeps track of performed tasks and manages all relevant metadata. The Cache also allows the user to inspect the data generated by a Recipe via the ASR command line interface (CLI) or using Python. Likewise, the Recipes may be executed directly from the CLI or called via Python scripts, the latter giving maximal flexibility and compatibility with existing Python libraries. For the purpose of high-throughput computations, advanced Python workflows combining several Recipes may be constructed and executed remotely using task scheduling systems like MyQueue [47].

The ASR Cache and Recipe modules work on a folder/file basis. This locality of data makes the ASR highly transparent for the user. The ASR Database module contains functions for converting the ASR data stored in a tree of folders into an ASE database and vice versa. The ASR App module generates web pages for online presentation, browsing and searching of the databases generated by the ASR Database module. Finally, the Data migration module provides tools for transforming data (results or metadata) to ensure backward compatibility when Recipes are updated.

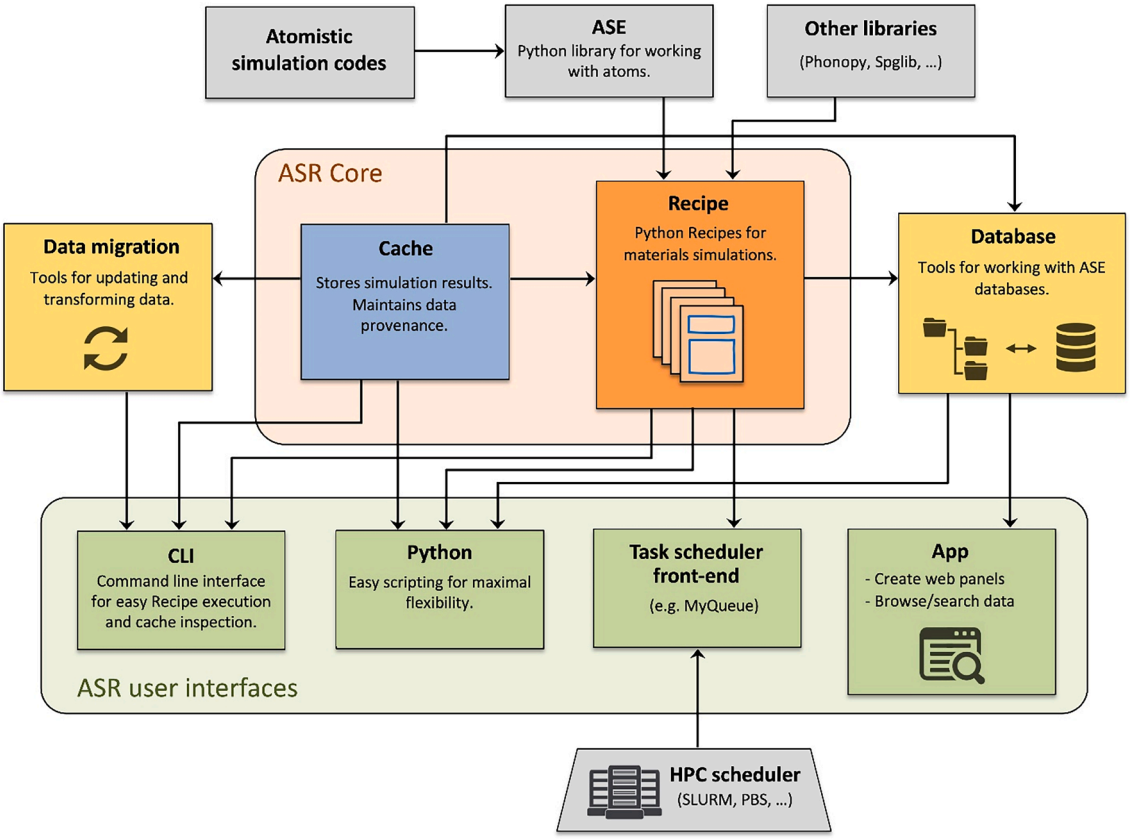

Fig. 1. Schematic overview of the main modules of the ASR and their interrelations. ASR consists of a Python library of Recipes for materials simulations and a caching system for recording of results and metadata. Recipes are envisioned to communicate with simulation codes via ASE interfaces, although most current Recipe implementations contain parts that are specific to the GPAW code. An arrow from X to Y means that Y calls X. The blue frames on the Instructions of the Recipe box symbolise a caching layer that records all data flow to/from the Recipes.

3. What is a Recipe?

A Recipe is a Python module implementing the Instructions needed to obtain a particular result, for example to relax an atomic structure, calculate an electronic band structure or a piezoelectric tensor. This section describes the structure and main components of a Recipe. A schematic overview of the Recipe concept is shown in Fig. 2.

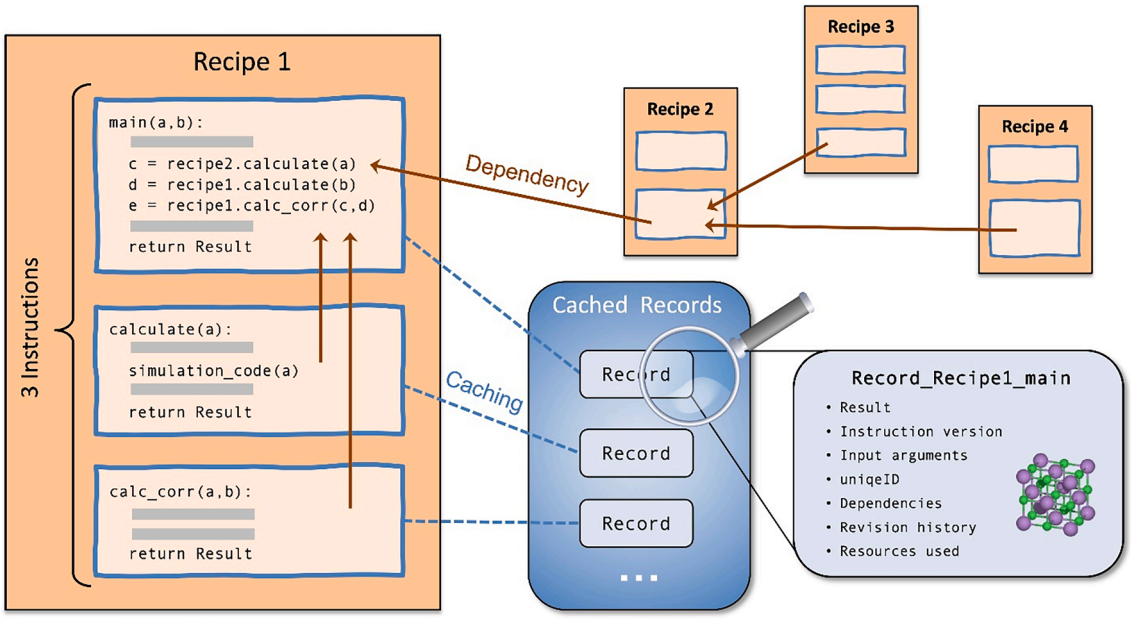

Fig. 2. A Recipe consists of a set of Instructions (see Fig. 3) implementing the computational steps needed to obtain a desired result. An Instruction may call other Instructions of the same, or separate, Recipes. An Instruction always returns a `Record` holding its result, normally represented as a `Result` data structure, together with the dependencies on other Instructions and all additional metadata required to trace back and reproduce the result.

3.1. Instruction

An Instruction is to be understood as a Python function wrapped in a caching layer provided by ASR, see Fig. 3. Whenever an Instruction is called, the caching layer intercepts the input arguments and asks the cache whether the result of the particular Instruction call already exists (cache hit) or whether there are no matching results (cache miss). If a matching result exists (because it was calculated previously), the caching layer skips the actual evaluation of the Instruction and simply reads and returns the previously calculated result. In the case of a cache miss, the Instruction is evaluated, after which the result is intercepted by the caching layer and stored together with the relevant metadata in a `Record` object. The precise content of the `Record` object and the conditions for a cache hit/miss are described in Section 3.3.

One of the great benefits of this design is its simplicity. Because the Instruction/caching layer is implemented as a simple wrapper around a Python function, usage of the caching functionality requires minimal additional knowledge. In practice, this means that working with ASR and implementing new ASR Recipes becomes really simple.

The caching system works on a per-folder basis (similar to Git): a cache is initialized by the user in a folder and any instruction evaluated within this folder or sub-folders will utilize this cache. This mimics the behaviour of the MyQueue task scheduler so as to maximize the synergy between these tools. In practice, the "one-cache-per-folder" system works well together with a "one-material-per-folder" structure. The latter is currently still a requirement for utilizing the Database functionalities described in Section 5. However, the caching system can work with several atomic structures in the same folder as the cache can distinguish ASR tasks performed on different atomic structures. Data written by ASR is encoded as `JSON`.

Any Instruction can be called directly by the user (from Python or the CLI), but special importance is given to the "main Instruction". The main Instruction usually provides the primary interface for the user to the Recipe and returns the final result of the Recipe. Other Instructions are called by the main Instruction and evaluated as needed. These may be Instructions implemented in the Recipe itself but may also be Instructions of separate Recipes. The main Instruction takes all input arguments required by the Recipe and uses them to call other Instructions.

Having multiple Instructions in a Recipe is usually motivated by code reusability or reduction of resources. The former is relevant when another Recipe needs to perform an identical Instruction (see Section on *Dependencies*). The latter is relevant when the task can be divided into Instructions with different resource requirements, in which case the separation may save computational time or resources. In particular, this is useful if a recalculation of a subset of the generated data is required.

The *input arguments* of an Instruction comprise all the information required to specify its task. When calls to an external simulation code are involved, the input arguments include a code specification, the computational parameters like k-point density, basis set specification, or exchange–correlation (xc) functional, as well the atomic structure.

An Instruction carries a *version number* to facilitate data migrations, i.e. transformations of the values or organisation of data produced by the Instruction. This may be required for backward compatibility when Instructions are updated, see Section 8.

3.2. Dependencies

It often happens that an Instruction can benefit from the functionality implemented by other Instructions. An example is the main Instruction of the "band structure" Recipe which calls an Instruction of the "ground state" Recipe to compute the electron density that the band structure should be based on. The caching layer logs whenever an Instruction requests data from another Instruction and uses that information to build a list of data dependencies. The data dependency list is stored in the `Record` object making it possible to trace what other pieces of data were used in the construction of the current result.

Implementation of data-dependencies in Recipes requires no extra coding. Whenever an Instruction calls another Instruction, the caching layer will *automatically* intercept the call and (1) determine if there exists a matching Record (cache hit/miss); (2) log the data dependency by registering the unique IDs and revision UIDS (see Section 8) of any dependent Records.

3.3. The `Record` object

The `Record` object is the basic data unit of ASR. It stores the results of Instructions together with metadata documenting how the results were obtained, and is used by the cache system to identify already performed Instruction calls. The `Record` object contains the following information:

- `Result` object (see Section 3.4)
- Input arguments, if relevant including
  - Atomic structure
  - Simulation code specification
  - Computational parameters
- Instruction version (see Section 8)
- External codes versions
- Randomly generated unique ID
- Dependencies (see Section 3.2)
- Revision History (see Section 8)
- Execution time and resources (number of cores)

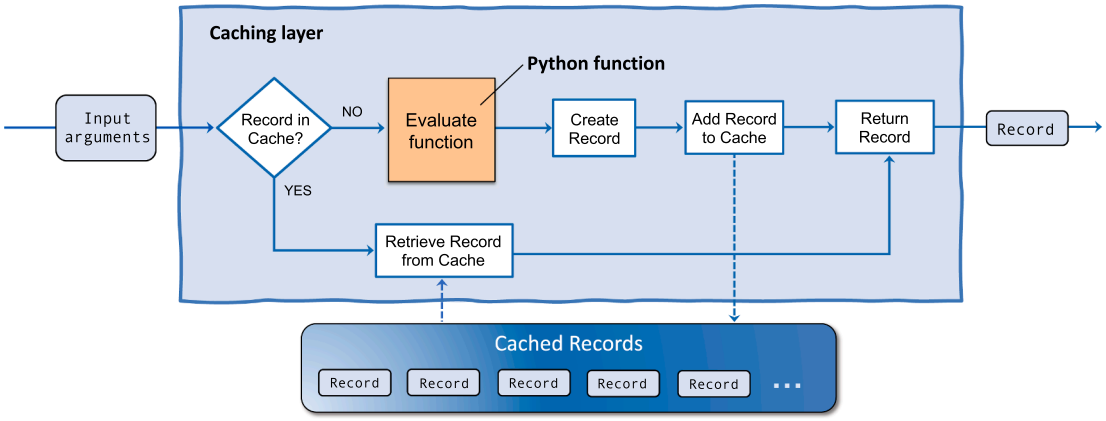

Fig. 3. An Instruction is a Python function (orange) wrapped in a caching layer (light blue). When the function is called with a set of input arguments, the caching layer consults the cache to check if a `Record` for that exact function call already exists. In case of a cache hit, the `Record` is read and returned. In case of a cache miss, the function is evaluated and the `Record` is stored before it is returned.

To identify a cache hit/miss when evaluating an Instruction, the caching layer searches the cache for `Records` with matching Instruction name, version, and input arguments. A cache hit is then defined as the existence of a matching `Record`. A recursive comparison is used to compare input arguments with those from existing `Records` within a small numerical tolerance for floating point numbers. Any later evaluations of the Instruction with identical arguments will result in a cache hit.

3.4. The `Result` object

To store and document the result produced by a Recipe, ASR offers a `Result` object that wraps the actual result data (stored as a Python dictionary) in a simple data structure that also contains specification of the result data types along with short explanatory descriptions of the data. In addition, the `Result` object may implement methods to present itself in different formats, see below. Using the `Result` object is optional, but in practice all Instructions that return more than a simple object or value utilizes a `Result` object for improved data documentation.

3.5. Presentation of results

The `Result` object may implement presentation options of the result data in various formats, for example text to terminal, figures, and web panels. The ASR Database and App modules draw on the Recipes' web panel implementation to create web pages for presenting, browsing, and distributing databases containing collected `Result` objects, see Sections 5 and 6. This provides an efficient way of inspecting and sharing data as it is generated, which is highly practical for projects involving multiple collaborators.

3.6. General principles for Recipe development

To maintain and exploit the modular structure of ASR, the development of new Recipes should follow a few general design principles. First, the task performed by a Recipe should be well defined and clearly bounded to make it easy to use in different contexts. It should always be considered whether the Recipe could be split into smaller independent Recipes that could be useful individually. Additionally, it is encouraged that Recipes are designed/programmed so as to be as broadly applicable as possible, e.g. with respect to the type of material (structure dimensionality, chemical composition, magnetic/non-magnetic, metallic/insulating, etc.). Any information required to define the simulation task should be included in the input argument of the Recipe, i.e. hard coding of parameters should be avoided. This should be done to ensure a flexible use and enhance the data provenance (input arguments are stored in the `Records`). Recipes should employ conservative parameter settings as default to ensure that the results are numerically well converged independent of the application, e.g. material type. Finally, in order to keep ASR Recipes simple and easy-to-read, and in order to enhance the modularity, code-extensive functionalities should be separated out into ASE functions and called from ASR whenever it is possible and sensible, i.e. when the ASE function is useful in other contexts than the specific Recipe.

4. The Recipe library

The ASR currently provides more than 40 complete Recipes allowing users to perform a broad range of materials simulation tasks ranging from construction and analysis of crystal structures over DFT calculations of thermodynamic, mechanical, electronic, magnetic, and optical properties to many-body methods for evaluating response functions, quasiparticle band structures, and collective excitations. A non-exhaustive list of available Recipes is provided in Table 1. It should be stressed that the list constitutes a snapshot of the current state of the Recipe library, which is continuously expanding. For example, we are currently developing Recipes for creating and modeling layered van der Waals structures and point defects in semiconductors.

Most of the currently implemented Recipes rely specifically on the GPAW [49] electronic structure code. As previously mentioned, we are currently working on a generalisation of the ASE Calculator interface to make the Recipes – or a large portion of them – simulation code-independent. Until then, usage of ASR with other simulation codes than GPAW is possible by porting of existing Recipes or development of new ones. The amount of work involved will depend on the type of Recipe and the state of the ASE interface for the specific simulation code.

A few specific examples of Recipe implementations are given in Section 6 where we outline the main computational steps and the final output of the `asr.bandstructure` and `asr.emasses` Recipes, respectively.

5. The ASR Database and App modules

The ASR Database and web App modules make it possible to package, inspect, share, and present ASR-driven projects easily and efficiently. The main tools and opportunities provided by these modules are described in more detail below.

5.1. Database

The ASR Database module can be used to collect `Record` objects from a directory tree into an ASE database. This is achieved by the command `asr database fromtree`. The procedure assumes a "one-material-per-folder" structure, relying on the existence of an atomic structure file in each folder to select `Records` pertaining to that atomic structure. This assumption, which is not fundamental, was chosen as a practical solution given the data layout of the computational 2D materials database. Alternative, more general solutions are being explored. The Database module proceeds to collect atomic structure-`Record` data sets and assign them to a particular row of an ASE database. We shall refer to such a database as an ASR database. Once an ASR database has been collected, it is possible to define key–value-pairs and relate property data to specific atomic structures.

The Database module also enables the reverse operation, that is, unpacking an existing ASR database to a directory tree containing `Record` objects. This is achieved by the command `asr database totree`. The function is useful when continuing a project, e.g. because existing data must be updated or new data must be added, for which the database is available but not the original directory tree. Moreover, the Database module provides tools for merging and splitting databases.

It is possible to collect a database for any number of materials/`Record` objects – even for a single material – and thereby take advantage of the App tools for presenting and inspecting results in a browser with no extra efforts. However, collecting databases is obviously most powerful in cases involving many materials/properties where the database makes it possible to search and filter the data via the defined key–value-pairs.

The easy installation of ASE through the standard PyPI Python package manager makes the ASE database format highly accessible. Furthermore, the portability of an ASE database (via several backends, e.g. SQLite, PostgreSQL, MariaDB and MySQL) enables easy packaging and distribution of data among different parties.

5.2. Web App

The ASE provides a flexible and easily extensible database web application making it possible to present and inspect the content of an ASE database in a browser. ASR leverages this ASE functionality to customize the web application layout and provide more sophisticated features such as the automatic generation of web panels, generation of figures, and documentation of the presented data by utilizing the web

panel data structures encoded in the `Result` objects. Normally a Recipe generates one web panel. However, panels gathering data from several Recipes may be created. One example of the latter is the "Summary" panel of the C2DB web pages discussed in the next section. In this case, a number of Recipes write data to a web panel data structure named "Summary" in their `Result` object. This information is stored in the database when collected. When generating the C2DB web pages from the C2DB database, the App constructs all web panels that are defined in the data pertaining to a particular material. If several Recipes have written to the same web panel, the data will be combined in an order controlled by a priority keyword written together with the web panel data.

5.2.1. Adding information fields

To enhance the accessibility of the data, it is possible to add an explanatory description to specific data entries, i.e, key–value pairs and data files, of an ASR database. These descriptions will appear as text boxes when clicking a "?"-icon placed next to the data on the web panels, see Fig. 5. General information boxes for web panels are always generated by ASR. They contain a customised field that can be manually edited, e.g. providing a short explanation of the data presented in the panel and/or links to relevant literature, and an automatically generated field listing the ASR Recipes that have produced data for the web panel and the key input parameters for the calculations. An example of such an information box is shown in Fig. 5.

5.2.2. Linking rows of databases

ASR provides functionality to create links between rows of the same, or different, ASR databases. This allows the developer to connect relevant materials when designing web panels such that the end user can move swiftly between them when browsing databases. For example, the `asr.convex_hull` Recipe creates the convex hull phase diagram of a material using an ASR reference database of stable materials (originally from the OQMD [28]), and creates a table with links to all the materials on the phase diagram. Other examples, could be to link different defective versions of the same crystalline material or different isomers of the same material/molecule.

The links are defined in `links.json` files in the folders of the relevant materials. These files may be generated manually or automatically using the Recipe `asr.database.treelinks`. When collecting the database, ASR reads the `links.json` file for each folder and stores the information in the `Data` dictionary of the corresponding row. The Recipe `asr.database.crosslinks` then creates links between rows of the collected database and rows of other databases that are given as input to the Recipe. When generating the web panels, ASR uses this information to generate hyperlinks in `HTML` format and present them in the web application for each material.

6. High-throughput example: The C2DB

In this section we present an example of what can be accomplished by the ASR in the realm of data intensive high-throughput applications, showcase some examples of ASR-generated web panels, and discuss two specific Recipe implementations.

Historically, the ASR evolved in a symbiotic relationship with the Computational 2D Materials Database (C2DB) — an extensive database project organising various properties of more than 4000 two-dimensional (2D) materials [32,48].

The C2DB distinguishes itself from existing computational databases of bulk [28–30] and low-dimensional [50,15,51] materials by the large number of physical properties available. These include convex hull diagrams, stiffness tensors, phonons (at high-symmetry points), projected density of states, electronic band structures with spin–orbit effects, effective masses, band topology indices, work functions, Fermi surfaces, plasma frequencies, magnetic anisotropies, magnetic exchange couplings, Bader charges, Born charges, infrared polarisabilities, optical absorption spectra, Raman spectra, and second harmonics generation spectra. The use of beyond-DFT theories for excited state properties (GW band structures and BSE absorption for selected materials) and Berry-phase techniques for band topology and polarization quantities (spontaneous polarization, Born charges, piezoelectric tensors), are other unique features of the C2DB.

Building the first version of C2DB without a fully functioning workflow framework was a long and painstaking endeavour, but absolutely critical for the successful development of the ASR. Today, the entire C2DB project can be generated by a single (MyQueue) Python workflow script comprising a sequence of ASR Recipe calls and simple Python code for controlling and directing the workflow via statements like "`if band_gap > 0:`". Relying on the MyQueue task scheduler (see Section 7.3), generation of the C2DB is accomplished by the single command "`mq workflow c2db_workflow.py tree/*/*/*/`", which will submit the C2DB workflow in folders matching the pattern `tree/*/*/*/`. With the current C2DB workflow, this statement will launch up to 23 unique Instructions for each of the 4047 materials amounting to a total of 59822 individual aiES calculations (some Recipes like phonon and stiffness calculations launch multiple aiES calculations). When the current workflow is run with the GPAW code, about 258 calculations are unsuccessful (most often due to convergence errors in the self-consistency DFT cycle) corresponding to a success rate of 99.5%.

Apart from the data provenance control that ensures the documentation and reproducibility of the data, there are two aspects of the ASR that are particularly crucial for making high-throughput computations work efficiently in practice. First, the caching functionality ensures that Recipes which have already been performed are automatically skipped by ASR (unless something in the input for a Recipe has changed since it was last executed). This means that only a single workflow script needs to be maintained and submitted every time something has been changed, e.g., new materials have been added, the workflow script has been updated, it has been decided to rerun certain tasks with new parameters, or a Recipe has been modified. Such functionality is essential because running and maintaining high-throughput projects inevitably requires that subsets of calculations are repeated at different points in time. Secondly, the carefully designed and well tested Recipes minimise the number of unsuccessful calculations and the risk of human errors.

6.1. Recipe and web page examples

Below we present a few examples of output generated by the ASR-C2DB workflow (for a full impression we refer the reader to the C2DB website).

6.1.1. Search page

Fig. 4 shows the C2DB search page, which consists of a search/filtering section followed by a list of the database rows presented by a selected number of key–value pairs. Clicking one of the highlighted key names once (twice) will sort the rows in increasing (decreasing) order of that key. Which keys should be shown by default can be customized, but the user can always add extra keys via the "Add column" button. By default, the search page generated by the ASR App module will contain only the search field in the upper section, but additional fields or buttons may be added for easy filtering according to the most relevant parameters.

6.1.2. "Summary" panel

Fig. 5 shows the C2DB web page for monolayer MoS_2. All the web panels produced by the various Recipes of the workflow are seen, but only the "Summary" panel is unfolded. This panel is designed to provide an overview of the most basic properties of the material, and gathers data from the `Result` objects generated by the following Recipes: `asr.gs`, `asr.gw`, `asr.hse`, `asr.phonons`, `asr.magstate`, `asr.stiffness`, `asr.convex_hull` and `asr.structureinfo`.

Fig. 5 also shows the information box of the "Effective masses" web

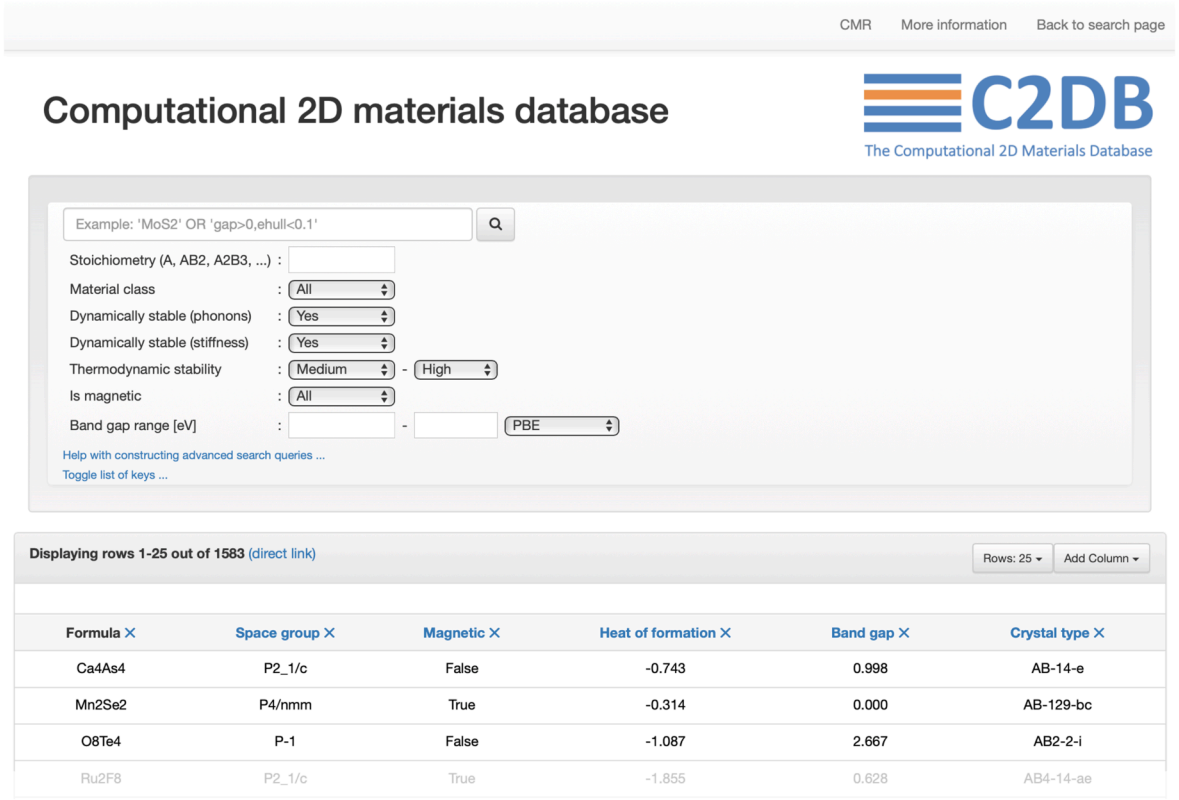

Fig. 4. The search page of C2DB with the first few rows of the database shown below. The default web page generated by ASR includes only the top most search field, but the panel can be customized by additional fields and buttons for more convenient data filtering.

panel. It contains a short explanation of the effective mass tensor and how it is evaluated by the Recipe as well as a link to a relevant paper. The automatically generated part shows that the panel contains data generated by the `asr.emasses` Recipe. The two fields at the top of the page "Download raw data" and "Browse raw data" provide access to the entire data set comprised by all `Result` objects of the specific material entry of the database.

6.1.3. *"Band structure" Recipe*

As another example, Fig. 6 shows the "Electronic band structure" panel for monolayer CrW_3S_8 as calculated and presented by the Recipe `asr.bandstructure`. The band structure is calculated with the PBE xc-functional including spin–orbit interactions. The out-of-plane spin projections of the states is shown by the color code. The main computational steps carried out by this Recipe are:

- Perform a self-consistent ground state calculation (by calling the `calculate` Instruction of the ground state Recipe `asr.gs`) to obtain a converged electron density.
- Determine crystal symmetries and corresponding band path (uses ASE functionalities).
- Calculate the Kohn–Sham eigenvalues along the band path. For magnetic materials, this step calls the Recipe `asr.magnetic_anisotropy` to obtain the magnetic easy axis for evaluating spin projections.
- Call the main Instruction of the ground state Recipe to get the Fermi level (in 3D) or the vacuum level (in < 3D) for use as zero-point energy for the band structure.

In addition to these computational steps, the main Instruction of the Recipe formats two figures to present the band structure itself and the Brillouin zone with the band path and the positions of the valence band maximum (VBM) and conduction band minimum (CBM). Note that the position of the VBM and CBM, as well as a number of other properties like the band gap and band edge energies (not shown), are determined by the Recipe `asr.gs`, which is called by `asr.bandstructure`.

6.1.4. *"Effective masses" Recipe*

Fig. 7 shows a screenshot of the "Effective masses" panel for monolayer CrW_3S_8 generated by the Recipe `asr.emasses`. The effective mass tensor is calculated with the PBE xc-functional including spin–orbit interactions. The color code represents the spin projections along the z-axis. In addition to the effective masses themselves, the Recipe evaluates a "band parabolicity" parameter defined as the mean absolute relative error (MARE) between the parabolic fit and the true bands in an energy range of 25 meV. The main computational steps carried out by this Recipe involve three subsequent k-point grid refinements; specifically:

- Perform a self-consistent ground state calculation on a uniform k-point grid (by calling the `calculate` Instruction of the Recipe `asr.gs`) to obtain a converged electron density as well as Kohn–Sham band energies.
- Locate the preliminary positions of the VBM and CBM and calculate band energies on a higher-density k-point grid around the VBM and CBM to locate the VBM and CBM positions with higher accuracy.
- Define final high-density k-point grids in the vicinity of the VBM and CBM points, and calculate band energies.
- Locate VBM and CBM and fit bands by second-order polynomial using band energies in an energy range of 1 meV from the band extremum.
- Calculate band structures for the web panel and evaluate the "parabolicity parameter".

It should be noted that even though effective mass calculations appear to be a simple task, it is surprisingly tricky to design a scheme that performs efficiently, robustly, and accurately across all types of

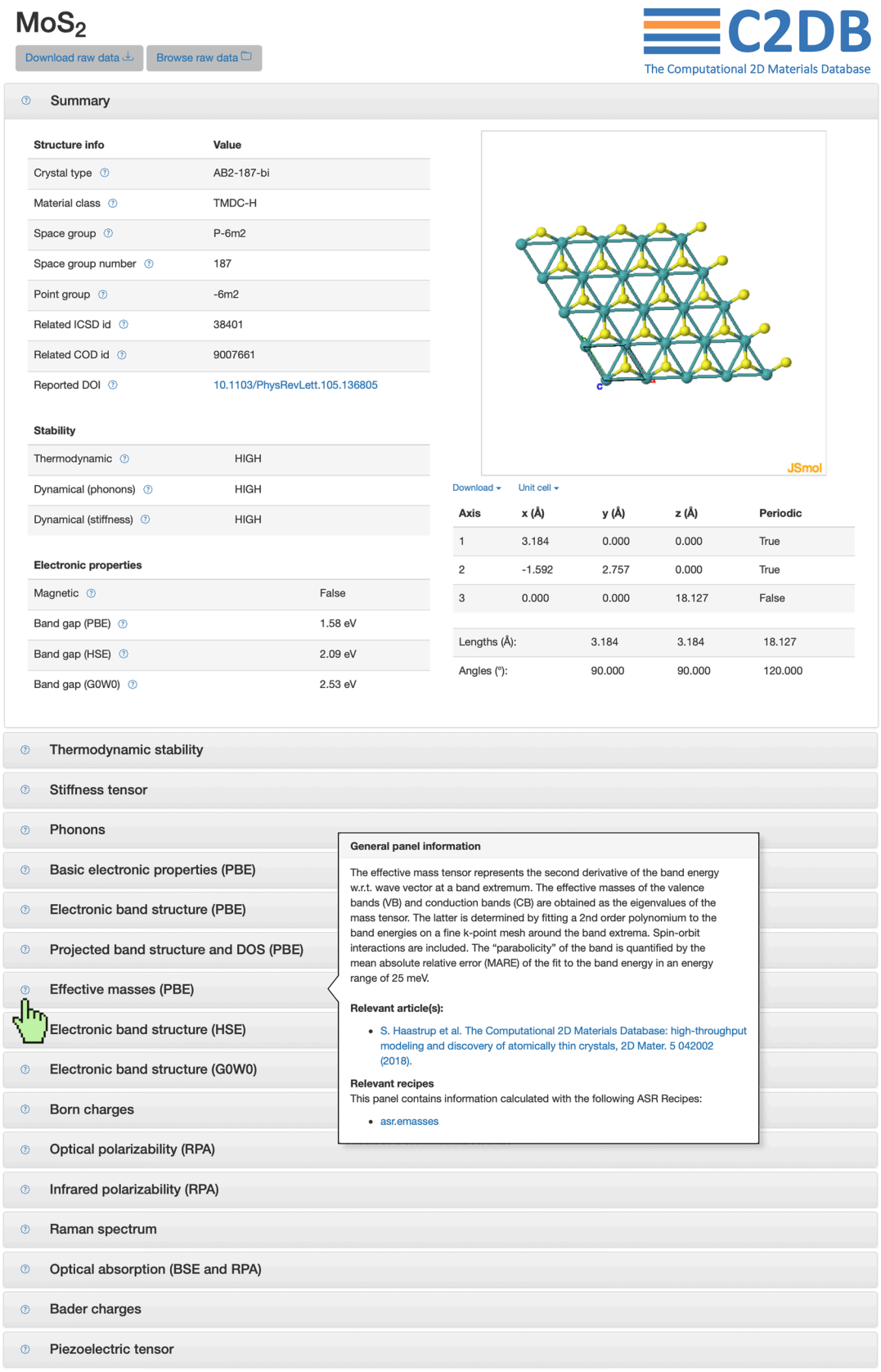

Fig. 5. Screenshot of the web page for monolayer MoS_2 from the C2DB project (only the "Summary" panel is unfolded). The panel presents data from the `Result` objects generated by the following Recipes: `asr.gs`, `asr.gw`, `asr.hse`, `asr.phonons`, `asr.magstate`, `asr.stiffness`, `asr.convex_hull`, `asr.structureinfo`.

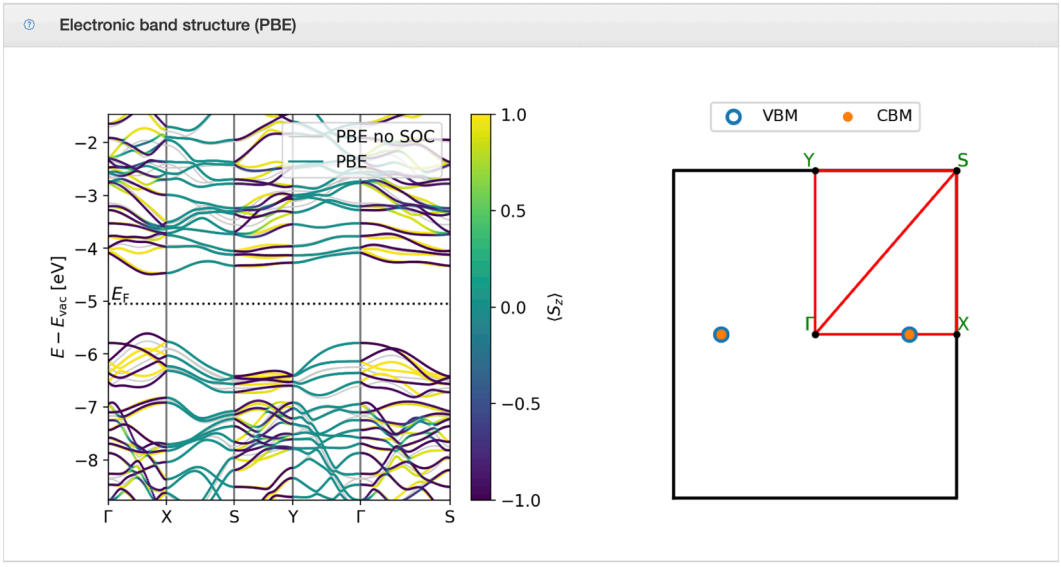

Fig. 6. Screenshot of the "Band structure" panel for monolayer CrW_3S_8 from the C2DB project. The web panel contains data computed by the `asr.bandstructure` Recipe.

Electron effective mass	Value	MARE (25 meV)
CB, direction 1	0.54 m_0	0.552 %
CB, direction 2	0.99 m_0	3.343 %
CB + 1, direction 1	0.63 m_0	0.77 %
CB + 1, direction 2	1.59 m_0	6.374 %

Hole effective mass	Value	MARE (25 meV)
VB, direction 1	0.49 m_0	0.006 %
VB, direction 2	0.48 m_0	0.011 %

Fig. 7. Screenshot of the "Effective masses" panel for monolayer CrW_3S_8 from the C2DB project. The panel contains data computed by the `asr.emasses` Recipe.

band structures including flat bands, highly dispersive bands, highly anisotropic bands, and bands exhibiting complex spin–orbit effects like Rashba splittings.

6.1.5. General comments

In contrast to the "Summary" panel, which has been customized for the C2DB project (that is, the web panel sections of the relevant Recipes have been appropriately adjusted), the "Electronic band structure" and "Effective masses" panels are the default web panels produced by the `asr.bandstructure` and `asr.emasses` Recipes, respectively.

The examples given here concern two-dimensional (2D) materials. However, the Recipes `asr.bandstructure` and `asr.emasses` (like all other Recipes of the current ASR library) apply also to 1D and 3D materials, as well as 0D where it is meaningful. As mentioned in Section 3.6, this kind of generality should always be strived for when designing Recipes. Achieving this may be straightforward or more involved depending on the Recipe. The Recipe for the stiffness tensor represents an easy case, where the dimensionality merely dictates the number of

axes along which the material must be strained. The Recipe for the band structure is more involved in this regard, as the determination of the band path requires separate treatments in 2D and 3D as does the determination of the spin projection axis (in 2D the out-of-plane direction is a natural choice while in 3D the magnetic easy axis is more appropriate).

7. User interfaces

The ASR can be used via four different interfaces, c.f. Fig. 1: A command line interface (CLI), a Python interface, a task scheduling front-end, and an app-based interface. Below we describe each interface in more detail.

7.1. The CLI

The CLI provides convenient commands for easy interaction with ASR via the `cache` and `run` subcommands. The `cache` subcommand allows inspection of the `Records` stored in the cache, in particular their `Result` data. For example, $ `asr cache ls name=asr.gs` will list all `Records` produced by the "ground state" Recipe. The `run` subcommand can be used to execute Recipes directly from the command line. For example, $ `asr run asr.gs` will run the *ground state* Recipe.

7.2. Python interface

The Python scripting interface allows inspection of `Records` and execution of Recipes directly from Python. This makes it possible to implement more complex logic and integrate directly with ASE and any other tools in the user's Python toolkit.

7.3. MyQueue interface

For high-throughput computations, ASR can be used in combination with a workflow manager that can handle the interaction with the scheduler of the supercomputer, such as Fireworks [40] or MyQueue [47]. The latter is a personal, decentralized, and lightweight front-end for schedulers (currently supporting SLURM, PBS, and LSF), which has been co-designed with ASR. MyQueue has a command line interface, which allows for submission of thousands of jobs in one command and provides easy-to-use tools for generating an overview of the status of jobs ('done', 'queued', 'failed' etc.). It also has a Python interface that can be used to define workflows. A Python script defines a dependency tree of *tasks* that MyQueue will submit without user involvement. The dependencies take the form: "if task X is done then submit task Y". MyQueue works directly with folders and files, which makes it transparent and easy to use. Together ASR and MyQueue provide a powerful and extremely flexible toolkit for high-throughput materials computations.

Individual Instructions of the Recipes may be defined as separate MyQueue tasks, such that computational resources can be specifically dedicated each Instruction ensuring a flexible and efficient execution of any workflow. It is, however, not a requirement to specify resources on a per Instruction basis, in which case the resources specified for the main Instruction will apply to all Instructions of the Recipe.

7.4. App interface

The App interface is a web-based read-only interface that allows the user to present and inspect the data stored in an ASE database on a local or public network. Distributing the data on a local network is convenient for larger projects and/or projects involving several users, as it allows for easy sharing and monitoring of the data as the project evolves. Once a project is finalized, the App may be used as a platform to present the data to the world via web pages. The data presentation used by the App is defined in the `Result` object of the Recipes.

8. Data maintenance

It sometimes happens that a Recipe, or one of its Instructions, has to be updated, e.g. because a bug has been detected or it has been found appropriate to store additional metadata. Such updates may imply that previously generated `Records` are no longer consistent with the current implementation of the Recipe. Depending on the nature of the change

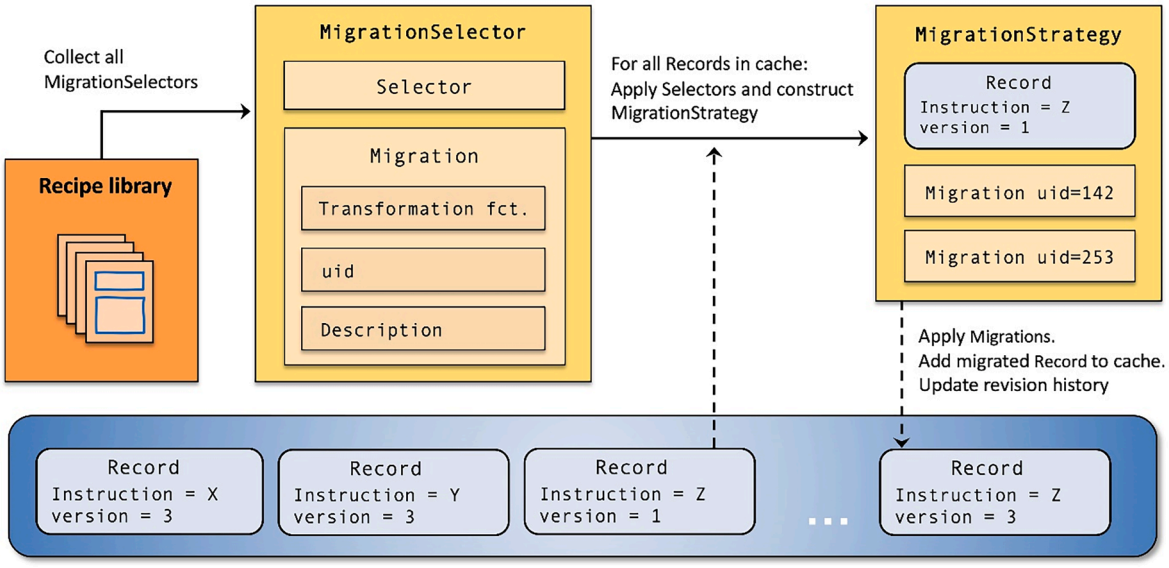

Fig. 8. To support data maintenance, ASR provides migration tools for bringing `Records` up-to-date with the latest version of the Instructions that produced them thereby avoiding recalculations whenever possible. The ASR migration procedure consists of the main steps: (1) Collect all `MigrationSelectors` from all available `Instructions`. (2) Select the migratable `Records` of the cache. (3) Determine a migration strategy (an ordered list of `Migrations`) for each migratable `Record`. (4) Apply transformation functions to migrate the `Records` and add them to the cache. (5) Update the revision history by a `Revision` object that documents the effect of the migration.

made to the Recipe, it may be possible to update the `Record` objects without rerunning the Recipe (data migration) or it may be necessary to rerun the entire Recipe or some of its Instructions (data regeneration).

To support the migration of data, ASR implements a simple versioning system for Instructions. An Instruction is associated with a integer version number which is stored in the `Record` and identifies the version of the Instruction at the time of creation. When an Instruction is changed, its version number may be increased by the developer. Since the caching layer matches the current Instruction version number against `Records` in the cache (see Section 3.3), older `Records` would no longer yield cache hits and are then said to be invalidated.

To facilitate the migration of invalidated `Records`, it is possible to specify `Migrations` that can be associated with an Instruction and thereby provide a way to bring old `Records` up to date. In practice, a `Migration` bundles a `Record` transformation function, a unique migration ID and a human readable description of the effect of the migration, see Fig. 8. In general, a `Record` transformation function induces a change to a `Record`. For example, this could be to convert a `Record` of version n to a later version $n+1$ without rerunning the Instruction, but in general the effect of the transformation could be anything. Use of transformation functions is typically possible when the update involves changes to metadata and/or data restructuring while the actual result of the Instruction is unchanged.

When a `Migration` is applied to a `Record`, a `Revision` object is produced. A `Revision` contains a randomly generated UID, the UID of the applied `Migration`, an explanatory description of the changes made to the `Record`, and an automatically generated list of the `Record` entries that were changed, added or deleted. The auto-generated list of changes is constructed by comparing the `Record` returned by the transformation function to the input `Record`.

Upon migration of a `Record`, a revision history is updated by the latest `Revision` and stored in the migrated `Record`. The revision history can be inspected by users to learn which revisions, if any, have previously been applied to a given `Record`.

A `Selector` is used to identify the `Records` to be migrated, e.g. based on the Instruction name and version number. The `Selector` is bundled together with a `Migration` into a `MigrationSelector`, which can determine whether a particular `Record` matches the selection criteria of the `Selector`. To migrate a `Record`, ASR searches through all Recipes to collect their `MigrationSelectors` (if they have any) and apply them to the `Record` to find a "migration strategy", i.e., which `Migrations` to be applied and in which order. The migration strategy is then encoded in a `MigrationStrategy`, which couples a particular `Record` to an ordered list of `Migrations`. The particular `MigrationStrategy` can then be applied to the Cache to execute the migration of the associated `Record`.

ASR provides a simple CLI, via `asr cache migrate`, to analyse existing `Records` in the cache, and identify migratable `Records`.

Whenever the `asr` version used in a given project is upgraded, a project participant should identify migratable `Records`, migrate them and then rerun the project workflow. Up-to-date `Records` will then be taken directly from the cache, whereas the Instructions with invalidated `Records` and no associated `Migration`, i.e. `Records` that the developer cannot migrate directly to the newest version, will be rerun.

In order to minimize the computational cost of bringing data up-to-date with ASR, developers are strongly advised to supply `Migrations` with their Recipe updates whenever possible.

To provide the best conditions for the long term deployment of ASR-generated data, the `asr` version of important projects should be upgraded regularly and the project workflow rerun. Obviously, this action may induce changes in the data. Whether this is acceptable or not is ultimately a strategic decision. However, for dynamic data projects, a regular version upgrade not only ensures that the data is of the highest quality, it also makes it easier for other parties to deploy the data because existing results (`Records`) can be reused directly with the newest version of ASR without having to rerun Recipes to bring the data up-to-date.

Table 2
Technical specifications.

Source code	https://gitlab.com/asr-dev/asr
Releases	https://pypi.org/project/asr/
License	GNU GPLv3 or newer (free software)
Documentation	https://asr.readthedocs.io/en/latest/

9. Data provenance

Simply stated, data provenance is the documentation of the circumstances under which a piece of data came into existence. This includes how the data originally was constructed, how the data has changed over time (also known as data-lineage) and a documentation of relevant system specifications such as architecture, operating system, important system packages, executables etc. If data provenance is handled perfectly, then data will in principle be reproducible, i.e. given access to exactly the same systems and software, any piece of data can be reproduced. In a scientific context, where reproducibility is key, data provenance is naturally very important.

In ASR, the basic unit of data is the `Record` object, which connects the result of an Instruction with various pieces of contextual metadata, see Section 3.3. Taken together, the metadata tell the story of how the original `Record` came into existence (Instruction name/version and input arguments), which other `Records` were implicitly used for the construction of this `Record` (Dependencies), what external package versions were used, and how the `Record` has transformed over time (Revision history). For simplicity, since it would be outside the scope of ASR, system information is not stored with the `Record`, which, in our experience, is not practically relevant for the purposes of ASR. As such, we characterize ASR as practically, but not perfectly, data provenant.

10. Documentation

ASR itself is documented on Read the Docs. The data is documented through the `Record` and `Result` objects, see previous Section on data provenance.

11. Technical specifications

Some technical specifications are listed in Table 2. ASR can be installed via pip using the command `pip install asr`.

ASR requires or is normally used with the following software:

- Python libraries: ASE, numpy, matplotlib, plotly, flask, click
- Computational and workflow software: GPAW or other ASE codes, MyQueue (SLURM/PBS/LSF)
- Optional extras: spglib, phonopy, and pymatgen (for Recipes); jinja, mysql or other ASE database backends

For community support see https://asr.readthedocs.io/en/latest/src/contact.html.

12. Summary and outlook

This article has introduced The Atomic Simulation Recipes (ASR) as an open source Python framework for developing materials simulation workflows and managing the data they produce.

To facilitate the transition to a paradigm of data-intensive science, ASR was designed to support the development of materials simulation workflows that operate in accordance with the FAIR data principles, by providing tools and concepts that are general enough that they do not restrict the user whilst being concrete enough to make a real difference. The ASR achieves this through the notion of a Recipe: a general Python

script that performs a well defined simulation task and is wrapped in a caching layer that logs all relevant metadata without involving the user. This construction places essentially no restrictions on the developer's freedom to design and control the workflow, but resolves the critical and complex issue of keeping track of the data provenance. We stress that the core of ASR, i.e. the Recipe concept and the caching system, is fully simulation code independent. In particular, it is not tied to materials simulations and could potentially be useful in other areas of computational science.

Beyond the built-in data documentation, there are many benefits of using standardized, well tested, and well documented Recipes. For example, it saves time and promotes a more sustainable scripting culture by reducing the need for individual researchers to write and maintain their own personal scripts (which can be hard for other to read and are often lost when the developer leaves the group). Furthermore, it reduces the risk of human errors and lowers the barrier for researchers to undertake simulation tasks with which they have little prior experience.

The fact that Recipes are independent units with own data provenance control implies that they can be freely combined to create advanced workflows using Python scripting for maximal flexibility. Such workflows can be executed on supercomputers using a workflow management software that supports a Python interface. To this end, we have developed the MyQueue [47] task manager that works as a front-end to the most common schedulers (currently SLURM, PBS, and LSF). While MyQueue will resubmit jobs that have timed out or crashed due to lack of memory, code-related failures must be handled manually. In the future, ASR should integrate more closely with MyQueue to permit that errors from the simulation codes are automatically analysed and reacted upon. Along the same lines, an automated estimation of the HPC resources (time/memory/nodes) required by individual tasks could limit the number of failed jobs and improve the utilization of resources.

The current Recipe library already covers a wide range of materials simulation tasks and more are continuously being added. Of special importance are Recipes for advanced beyond-DFT calculations where the benefits in terms of a lowered user barrier, improved data quality, and increased utilization of computing resources, are particularly large. The Recipe concept should also be advantageous for implementation of machine learning methods that could integrate with ASR databases and "standard" Recipes to make for more intelligent and computationally efficient workflows.

The ASR makes extensive use of the Atomic Simulation Environment (ASE) as a toolkit to process atomistic calculations. In particular, ASE is used as a front-end for ASR to communicate with external simulation codes. This has the clear advantage that ASR can become decoupled from the simulation codes. This decoupling is currently not in place, and the majority of the existing Recipe implementations contain code parts that are specific to the GPAW electronic structure code. To make ASR fully simulation code-independent, the ASE Calculator interfaces must be further generalized. This includes extensions of the interfaces to access outputs of calculations as well as a systematic mechanism to control multi-step tasks. The adaptation of this interface to multiple codes will eventually require a community effort that we hope many code developers will take part in. Until then, Recipes must to some extent be code specific.

CRediT authorship contribution statement

Morten Gjerding: Conceptualization, Methodology, Software, Validation, Writing - original draft, Writing - review & editing, Visualization. **Thorbjørn Skovhus:** Conceptualization, Software, Validation, Writing - review & editing, Visualization. **Asbjørn Rasmussen:** Conceptualization, Software, Validation, Writing - review & editing. **Fabian Bertoldo:** Conceptualization, Software, Validation, Writing - review & editing. **Ask Hjorth Larsen:** Conceptualization, Software, Validation, Writing - review & editing. **Jens Jørgen Mortensen:** Conceptualization, Software, Validation, Writing - review & editing. **Kristian Sommer Thygesen:** Conceptualization, Validation, Writing - original draft, Writing - review & editing, Visualization, Funding acquisition, Project administration.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We acknowledge funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program Grant No. 773122 (LIMA) and Grant agreement No. 951786 (NOMAD CoE).

References

[1] J. Greeley, T.F. Jaramillo, J. Bonde, I. Chorkendorff, J.K. Nørskov, Nat. Mater. 5 (2006) 909–913.
[2] G.K. Madsen, J. Am. Chem. Soc. 128 (2006) 12140–12146.
[3] S. Curtarolo, et al., Nat. Mater. 12 (2013) 191–201.
[4] S. Kirklin, B. Meredig, C. Wolverton, Adv. Energy Mater. 3 (2013) 252–262.
[5] K.B. Ørnsø, J.M. Garcia-Lastra, K.S. Thygesen, Phys. Chem. Chem. Phys. 15 (2013) 19478–19486.
[6] Z. Zhang, et al., ACS Omega 4 (2019) 7822–7828.
[7] W. Chen, et al., J. Mater. Chem. C 4 (2016) 4414–4426.
[8] J. Hachmann, R. Olivares-Amaya, S. Atahan-Evrenk, C. Amador-Bedolla, R. S. Sánchez-Carrera, A. Gold-Parker, L. Vogt, A.M. Brockway, A. Aspuru-Guzik, J. Phys. Chem. Lett. 2 (2011) 2241–2251.
[9] S. Bhattacharya, G.K. Madsen, Phys. Rev. B Condens. Matter 92 (2015), 085205.
[10] I.E. Castelli, et al., Energy Environ. Sci. 5 (2012) 5814–5819.
[11] G. Hautier, A. Miglio, G. Ceder, G.M. Rignanese, X. Gonze, Nat. Commun. 4 (2013) 1–7.
[12] L. Yu, A. Zunger, Phys. Rev. Lett. 108 (2012), 068701.
[13] K. Kuhar, M. Pandey, K.S. Thygesen, K.W. Jacobsen, ACS Energy Lett. 3 (2018) 436–446.
[14] M. Aykol, S. Kim, V.I. Hegde, D. Snydacker, Z. Lu, S. Hao, S. Kirklin, D. Morgan, C. Wolverton, Nat. Commun. 7 (2016) 1–12.
[15] N. Mounet, M. Gibertini, P. Schwaller, D. Campi, A. Merkys, A. Marrazzo, T. Sohier, I.E. Castelli, A. Cepellotti, G. Pizzi, et al., Nat. Nanotechnol. 13 (2018) 246–252.
[16] L.Q. Chen, L.D. Chen, S.V. Kalinin, G. Klimeck, S.K. Kumar, J. Neugebauer, I. Terasaki, NPJ Comput. Mater. 1 (2015) 1–2.
[17] M. Rupp, A. Tkatchenko, K.R. Müller, O.A. Von Lilienfeld, Phys. Rev. Lett. 108 (2012), 058301.
[18] J. Lee, A. Seko, K. Shitara, K. Nakayama, I. Tanaka, Phys. Rev. B 93 (2016), 115104.
[19] T. Xie, J.C. Grossman, Phys. Rev. Lett. 120 (2018), 145301.
[20] L.M. Ghiringhelli, J. Vybiral, S.V. Levchenko, C. Draxl, M. Scheffler, Phys. Rev. Lett. 114 (2015) 105503.
[21] P.B. Jørgensen, E.G. del Río, M.N. Schmidt, K.W. Jacobsen, Phys. Rev. B 100 (2019) 104114.
[22] K. Ghosh, A. Stuke, M. Todorović, P.B. Jørgensen, M.N. Schmidt, A. Vehtari, P. Rinke, Adv. Sci. 6 (2019) 1801367.
[23] V.L. Deringer, G. Csányi, Phys. Rev. B 95 (2017), 094203.
[24] S. Lorenz, A. Groß, M. Scheffler, Chem. Phys. Lett. 395 (2004) 210–215.
[25] J. Behler, M. Parrinello, Phys. Rev. Lett. 98 (2007), 146401.
[26] N. Artrith, A. Urban, Comput. Mater. Sci. 114 (2016) 135–150.
[27] K.S. Thygesen, K.W. Jacobsen, Science 354 (2016) 180–181.
[28] J.E. Saal, S. Kirklin, M. Aykol, B. Meredig, C. Wolverton, JOM 65 (2013) 1501–1509.
[29] A. Jain, et al., APL Mater. 1 (2013), 011002.
[30] S. Curtarolo, W. Setyawan, G.L. Hart, M. Jahnatek, R.V. Chepulskii, R.H. Taylor, S. Wang, J. Xue, K. Yang, O. Levy, et al., Comput. Mater. Sci. 58 (2012) 218–226.
[31] C. Draxl, M. Scheffler, J. Phys.: Mater. 2 (2019), 036001.
[32] S. Haastrup, M. Strange, M. Pandey, T. Deilmann, P.S. Schmidt, N.F. Hinsche, M.N. Gjerding, D. Torelli, P.M. Larsen, A.C. Riis-Jensen, et al., 2D Mater 5 (2018) 042002.
[33] S.S. Borysov, R.M. Geilhufe, A.V. Balatsky, PloS One 12 (2017), e0171501.
[34] K.T. Winther, M.J. Hoffmann, J.R. Boes, O. Mamun, M. Bajdich, T. Bligaard, Sci. Data 6 (2019) 1–10.
[35] L. Talirz, S. Kumbhar, E. Passaro, A.V. Yakutovich, V. Granata, F. Gargiulo, M. Borelli, M. Uhrin, S.P. Huber, S. Zoupanos, et al., Sci. Data 7 (2020) 1–12.
[36] R. Armiento, Mach. Learn. Meets Quant. Phys. (2020) 377–395.
[37] L. Himanen, A. Geurts, A.S. Foster, P. Rinke, Adv. Sci. 6 (2019) 1900808.
[38] W. Kohn, L.J. Sham, Phys. Rev. 140 (1965) A1133.
[39] M.D. Wilkinson, M. Dumontier, I.J. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J.W. Boiten, Santos L.B. da Silva, P.E. Bourne, et al., Sci. Data 3 (2016) 1–9.

[40] A. Jain, et al., Concurr. Comput. 27 (2015) 5037–5059.
[41] G. Pizzi, A. Cepellotti, R. Sabatini, N. Marzari, B. Kozinsky, Comput. Mater. Sci. 111 (2016) 218–230.
[42] K. Mathew, J.H. Montoya, A. Faghaninia, S. Dwarakanath, M. Aykol, H. Tang, I.h. Chu, T. Smidt, B. Bocklund, M. Horton, et al., Comput. Mater. Sci. 139 (2017) 140–152.
[43] A.H. Larsen, J.J. Mortensen, J. Blomqvist, I.E. Castelli, R. Christensen, M. Dułak, J. Friis, M.N. Groves, B. Hammer, C. Hargus, et al., J. Phys.: Condens. Mat. 29 (2017), 273002.
[44] S.P. Ong, W.D. Richards, A. Jain, G. Hautier, M. Kocher, S. Cholia, D. Gunter, V. L. Chevrier, K.A. Persson, G. Ceder, Comput. Mater. Sci. 68 (2013) 314–319.
[45] A. Togo, I. Tanaka, 2018 arXiv preprint arXiv:1808.01590.
[46] A. Togo, L. Chaput, I. Tanaka, Phys. Rev. B 91 (2015), 094306.
[47] J. Mortensen, M. Gjerding, K. Thygesen, J. Open Sour. Softw. 5 (2020) 1844.
[48] M. Gjerding, et al., 2D Mater 8 (2021), 044002.
[49] J. Enkovaara, C. Rostgaard, J.J. Mortensen, J. Chen, M. Dułak, L. Ferrighi, J. Gavnholt, C. Glinsvad, V. Haikola, H. Hansen, et al., J. Phys.: Condens. Mat. 22 (2010), 253202.
[50] M. Ashton, J. Paul, S.B. Sinnott, R.G. Hennig, Phys. Rev. Lett. 118 (2017), 106101.
[51] J. Zhou, L. Shen, M.D. Costa, K.A. Persson, S.P. Ong, P. Huck, Y. Lu, X. Ma, Y. Chen, H. Tang, et al., Sci. Data 6 (2019) 1–10.