Contents lists available at ScienceDirect

SoftwareX

Original software publication

Simplifying computational workflows with the Multiscale Atomic Zeolite Simulation Environment (MAZE)

Dexter D. Antonio, Jiawei Guo, Sam J. Holton, Ambarish R. Kulkarni *

University of California, Davis Department of Chemical Engineering, United States of America

ARTICLE INFO

ABSTRACT

Zeolites, an important class of 3-dimensional nanoporous materials, have been widely explored for a variety of applications including gas storage, separations, and catalysis. As the properties of these aluminosilicate materials depend on a number of factors (e.g., framework topology, Si/Al ratio, extra-framework cations etc.), detailed experiments (e.g., catalytic properties, adsorption capacities etc.) are often limited to only a handful of materials. Computational methods have played an important role in (1) providing molecular level insights to rationalize experimental observations, and (2) screening large libraries of zeolites to identify promising candidates for experimental synthesis and validation. Different levels of theory and computational chemistry codes are necessary to describe the range of relevant phenomena such as adsorption (e.g., grand canonical Monte Carlo), diffusion (e.g., molecular dynamics), and chemical reactions (e.g., density functional theory). Manipulation of atomic structures, handling of input files, and developing robust workflows becomes quite cumbersome. To mitigate these challenges, we describe the development of the Multiscale Atomic Zeolite Simulation Environment (MAZE) – a Python package that simplifies zeolite-specific calculation workflows by providing a user-friendly interface for systematically manipulating zeolite structures.

Code metadata

Current Code version	v0.1.1
Permanent link to code/repository used of this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-21-00107
Legal Code License	MIT
Code Versioning system used	git
Software Code Language used	python
Compilation requirements, Operating environments & dependencies	Python Packages: ase, numpy, typing, packaging, scikit-learn
If available Link to developer documentation/manual	https://kul-group.github.io/MAZE-sim/
Support email for questions	Dexter.d.antonio@gmail.com

Software metadata

Current software version	v0.1.1
Permanent link to executables of this version	https://github.com/kul-group/MAZE-sim/releases/tag/0.1.0
Legal Software License	MIT
Computing platform/Operating System	Linux, OS X, Microsoft Windows, Unix-like
Installation requirements & dependencies	Python3 Python Packages: ase, numpy, typing, packaging, scikit-learn
If available Link to user manual - if formally published include a reference to the publication in the reference list	https://kul-group.github.io/MAZE-sim/
Support email for questions	Dexter.d.antonio@gmail.com

1. Introduction

Zeolites are a broad class of silica-based nanoporous materials which are widely used for various industrial applications

* Correspondence to: Department of Chemical Engineering, 1 Shields Ave Davis, CA 95616, United States of America.
E-mail address: arkulkarni@ucdavis.edu (Ambarish R. Kulkarni).

including gas separation and catalysis [1–3]. Transition metal (TM) exchanged zeolites combine the desirable characteristics of heterogeneous catalysts (high thermal stability and simpler separations) with those of enzymes (high selectivity and reactivity under mild conditions) [3] and have received considerable attention as catalysts for numerous reactions, such as NOx abatement [4] and methane valorization [5]. Computational modeling is often used to provide insights (e.g., thermodynamic stabilities [6], reaction barriers [7] etc.) into the reaction mechanisms and properties of zeolites [8,9]. Given the various length- and time-scales associated with molecular processes (e.g., adsorption, diffusion, reaction), multiscale approaches that combine wave function theory, periodic density functional theory and classical force fields are often necessary [8]. While a number of broadly-applicable software packages are available for performing these calculations [10], a software toolkit for zeolite-specific tasks would be valuable. In this work, we describe the design and capabilities of a new python-based open-source software package — Multiscale Atomistic Zeolite Simulation Environment (MAZE).

The increasing availability of open-source software packages [11] that offer an user-friendly interfaces has greatly simplified the process of performing computational chemistry calculations. For example, the Atomic Simulation Environment (ASE), provides interfaces to various computational chemistry codes (e.g., VASP [12], LAMMPS [13], GPAW [14]). ASE provides Python-based wrappers to the underlying quantum chemical simulation code and offers an intuitive application programming interface (API) for setting up, starting, and analyzing calculations [15,16]. By automating the cumbersome computational setups and subsequent data analysis, ASE simplifies the process of performing and analyzing complex calculations [15,17]. Furthermore, by allowing manipulation through Python scripts, rather than a GUI, these calculations become self-documenting, reproducible and easy to streamline into complex workflows [11,15].

2. Problems and background

2.1. Current limitations with tracking atoms within the ASE interface

Despite an active user community and continued developments within the ASE codebase [15], a few specific structural manipulation tasks are challenging to implement within ASE. Often a variety of structural manipulations (e.g., extracting and reinserting clusters, adding terminal H atoms etc.) are necessary to address a zeolite-specific scientific question—the current ASE interface is not well suited for "tracking" the resulting changes in the atom indices. This is illustrated using a simple example below.

In ASE, groups of atoms are represented in memory by Atoms Python objects. In an Atoms object, the properties of all of the atoms are stored in NumPy arrays. When a specific atom in an Atoms object is accessed using the get_item method (e.g., my_atoms[index]) an Atom object is created, which has (among others) the attributes 'tag', 'position', 'symbol' and 'index'. The underlying data structure for storing the atoms properties is highly efficient, since it does not require storing an individual Python object for each atom represented by an Atoms object. Unfortunately, it also means that the indices of each atom can change with each addition or deletion. Fig. 1 shows the structures and atom indices for various glyoxal derivatives, demonstrating how structural manipulations can alter the indices of atoms.

Fig. 1 shows how the indices of the individual atoms can change when additions and deletions are performed. The most pronounced difference is in the fourth structure, where the deletion of two atoms causes the decrement of the index 4 and 5 to 2 and 3 respectively. The addition of atoms simply extends the arrays and are thus added to the end. There is no inherent

ordering in the indices; the order in which atoms are added to a structure effects the final order of atoms. If a substitution is made by changing the identity of a given atom, then the indices remain unchanged. The mutability of Atoms objects, and subsequent index shifting, introduces significant complexity in tracking the relationship and identity of atoms. These are a major bottlenecks for developing workflows that include both periodic and cluster calculations with zeolites.

2.2. The MAZE solution

Recognizing the challenges outlined above, the MAZE package puts the atom relationship tracking at the center of its design, while maintaining compatibility with all existing ASE's features. As demonstrated in the following sections, it greatly simplifies zeolite workflows and reduces the difficulty in performing a series of structural manipulations.

3. MAZE architecture

3.1. Architecture overview

The MAZE project aims to include all of the functionality of the base ASE package while including additional functionality related to the tracking of atoms. This is incorporated by using inheritance. A zeolite is a group of atoms, so it is appropriate to create a Zeolite class that inherits from ASE's Atoms class. The Zeolite class represents a zeolite and includes additional methods and properties for identifying the unique crystallographic sites. Polymorphism ensures that the Zeolite class has all of the attributes and methods of the parent Atoms class. Thus, all of ASE's methods and classes also work well with it.

The additional functionality of the Zeolite class is divided between two classes, the parent PerfectZeolite class and its subclass Zeolite. The PerfectZeolite class includes the functionality for building a Zeolite from a labeled CIF file and preserving the site labels. The methods included in the PerfectZeolite are all of those related to site identification, and serialization. In a group of zeolites there can be only one perfect zeolite, from which all the derivatives (e.g., Bronsted H versions, adsorbates etc.) are made. A simplified unified modeling language (UML) class diagram for the Zeolite and PerfectZeolite classes is presented in Fig. 2.

Users of the MAZE package will interact primarily with Zeolite objects. The main additional features of the Zeolite class versus the PerfectZeolite class are related to atom manipulation, such as adding atoms, deleting atoms, extracting clusters and capping clusters. By dividing the functionality between two classes, the attributes that make a Zeolite and those involved in structural manipulation can be separated, greatly simplifying the underlying code. The underpinning of the Zeolite functionality is an internal IndexMapper object, which tracks the relationship between the indices of the atoms in the zeolites derived from the same parent structure.

3.2. The IndexMapper class

The instances of the IndexMapper class are responsible for tracking the relationship between atom indices. A reference to an IndexMapper object is an attribute of each Zeolite class and related Zeolites share the same IndexMapper. The IndexMapper does not directly encounter Atoms objects, but only works with their indices. The core data structure of the IndexMapper is the main_index, which consists of a collection of nested dictionaries. The key of the outer dictionary is the unique id of each row of atoms in the object (Fig. 3). The inner dictionary consists of each

Structure	Transformation	Indices of the NumPy Array							
	None	0	1	2	3	4	5	None	None
	Add Two H Atoms	0	1	2	3	4	5	6	7
	Replace the O Atom by S	0	1	2	3	4	5	None	None
	Delete the O and H on C1	0	1	None	None	2	3	None	None

Fig. 1. Relationship between indices in Atoms objects derived from glyoxal. The columns relate the indices of different Atoms objects to each other and the atoms in each structure are labeled with their corresponding indices. The indices shifting issue becomes more pronounced when multiple structural operations are performed in series. For example, if the O and H atoms are added back to #3 to recover #1, their indices (4 and 5) would differ from the initial structure.
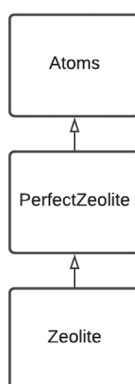

Fig. 2. Simplified unified modeling language (UML) class diagram for the Zeolite object. Inheritance relationships are denoted by an open arrow. The Zeolite class inherits from the PerfectZeolite class, which inherits from the Atoms class.

zeolites name attribute followed by the index of an Atom or a None object.

The main_index in the IndexMapper object records the relationship indices of different Atoms objects. The unique_id assigns a unique identifier (ID) to each atom. This ID does not depend on the atom species and if an atoms type is changed from silicon to tin, for example, the ID remains unchanged. The row shows the relationship between the indices across different Atoms objects. For example, in row four (i.e., index 3 of the dictionary), the ID equals 3 and the indices of the parent, Zeolite_1 and Open Defect_5 are all equal to 3. The equivalent atom index in Cluster_3, which consists of an extracted cluster from Zeolite_1 is 0. Cluster_3 consists of 21 atoms, and the Open Defect_5 object consists of all of the atoms in Zeolite_1 with the exception of the atoms in Cluster_3. Thus, Open Defect_5 final indices are offset by 21 as can be seen in the final rows of the table.

The main_index is automatically updated when each atom manipulation operation is performed and does not require additional intervention from the user. The index mapper class can be used to directly map between two related zeolites with the get_index function, yet its core benefit comes about by enabling the structural manipulation functions such as cap_atoms and integrate.

Complimenting this index mapper are the add_atoms and delete_atoms methods in the Zeolite class, which return a copy of the original Zeolite object with the applied modifications and append a new column to the IndexMapper's main index. This new column contains the indices of the newly created Zeolite object and each row indicates the relationship between the atoms in other zeolites. If a Zeolite object is deleted, then the deconstructor

```
{0: {"parent": 0, "Zeolite_1": 0, "Cluster_3": None, "Open Defect_5": 0},
 1: {"parent": 1, "Zeolite_1": 1, "Cluster_3": None, "Open Defect_5": 1},
 2: {"parent": 2, "Zeolite_1": 2, "Cluster_3": 0, "Open Defect_5": None},
 3: {"parent": 3, "Zeolite_1": 3, "Cluster_3": None, "Open Defect_5": 2},
 4: {"parent": 4, "Zeolite_1": 4, "Cluster_3": None, "Open Defect_5": 3},
 # ...
 188: {"parent": 188, "Zeolite_1": 188, "Cluster_3": None, "Open Defect_5": 167},
 189: {"parent": 189, "Zeolite_1": 189, "Cluster_3": None, "Open Defect_5": 168},
 190: {"parent": 190, "Zeolite_1": 190, "Cluster_3": None, "Open Defect_5": 169},
 191: {"parent": 191, "Zeolite_1": 191, "Cluster_3": None, "Open Defect_5": 170}}
```

Fig. 3. Dictionary representation of the main index mapper for a collection of related Zeolites. The keys of the outer dictionary represent the unique IDs, where the inner dictionaries map the relationship between indices for the same shared atom across different Atoms-like objects.
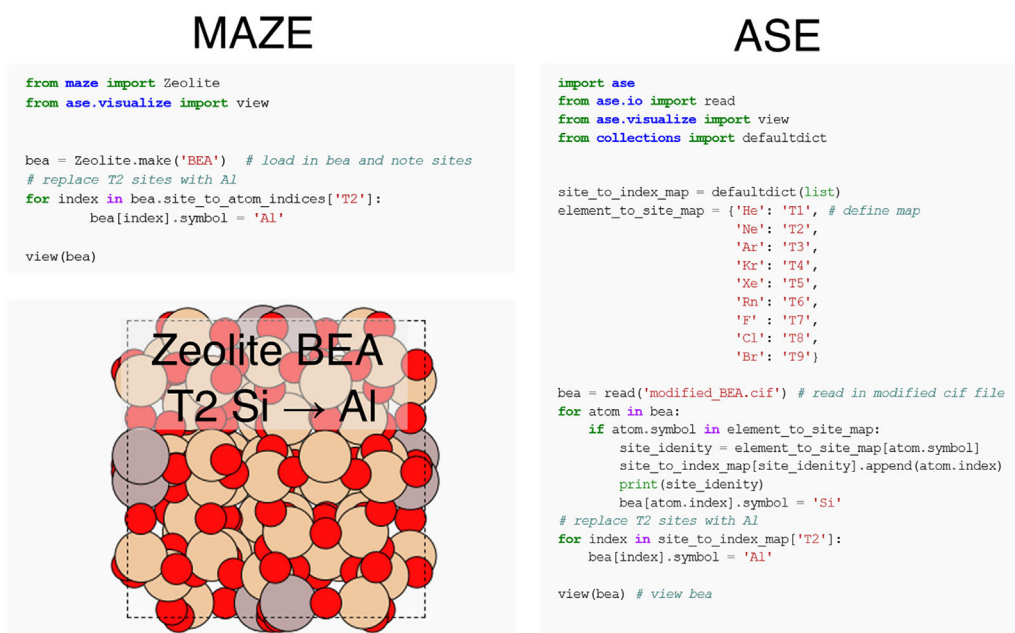
MAZE

```python
from maze import Zeolite
from ase.visualize import view


bea = Zeolite.make('BEA')  # load in bea and note sites
# replace T2 sites with Al
for index in bea.site_to_atom_indices['T2']:
        bea[index].symbol = 'Al'

view(bea)
```


ASE

```python
import ase
from ase.io import read
from ase.visualize import view
from collections import defaultdict


site_to_index_map = defaultdict(list)
element_to_site_map = {'He': 'T1',  # define map
                       'Ne': 'T2',
                       'Ar': 'T3',
                       'Kr': 'T4',
                       'Xe': 'T5',
                       'Rn': 'T6',
                       'F' : 'T7',
                       'Cl': 'T8',
                       'Br': 'T9'}

bea = read('modified_BEA.cif')  # read in modified cif file
for atom in bea:
    if atom.symbol in element_to_site_map:
        site_identity = element_to_site_map[atom.symbol]
        site_to_index_map[site_identity].append(atom.index)
        print(site_idenity)
        bea[atom.index].symbol = 'Si'
# replace T2 sites with Al
for index in site_to_index_map['T2']:
    bea[index].symbol = 'Al'

view(bea)  # view bea
```

Fig. 4. Comparison between MAZE and ASE code for generating a BEA zeolite structure with the Silicon T2 sites replaced by aluminum atoms. The MAZE code uses the built-in make function to read the unmodified CIF file and store the mapping in the site_to_atom_indices dictionary. The longer ASE code requires a modified CIF file as input, and the element mapping to be manually defined. Both codes generate and visualize the same BEA T2 Si→Al structure.
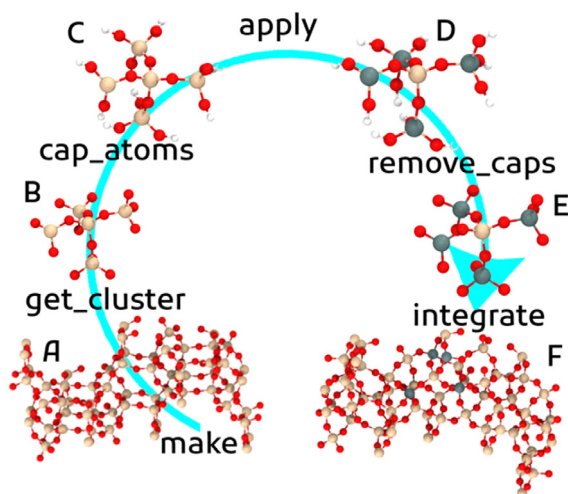

Fig. 5. Workflow for cluster integration. A blue arrow shows the workflow direction, starting with a BEA zeolite constructed using the make method. The functions or operations required to transfer from one structure to another are shown between the structures on the blue line.

will remove its corresponding entry from the IndexMapper, preventing the main index from being cluttered with deleted Zeolite object indices.

4. Illustrative examples

To demonstrate the capability of the MAZE code and assess its API three distinct tasks were performed. These tasks include building a Zeolite object from a labeled International Zeolite Association (IZA) CIF file, adding and removing atoms, and a complete workflow involving removing a cluster, changing some of its atoms and reinserting it back into the original zeolite.

4.1. Building a zeolite from a CIF file

Zeolites often contain multiple distinct T-sites, each of which has unique chemistries arising from differences in the local atomic environment. Comprehensive zeolite screenings studies require all unique T sites to be systematically explored. Computational studies of this type start by downloading a CIF file from the IZA database, placing the file in the project folder and reading the CIF file into an Atoms object with the ase.io.read function. One challenge with this approach is that CIF files downloaded from the IZA database contain extra information about the identity of unique atoms, which is not preserved when the CIF file is loaded with ASE's read function. Thus, various "hacks" are needed to align the Atoms object built from the CIF file with their labels. One hack used in our group involves changing a unique site from a silicon to an unused atom such as xenon, and when the Atoms object is loaded reverting it back to a silicon, noting the indices (see Fig. 4, right). This manual tagging mechanism is slow, opaque because the code is no longer self-documenting, and error prone, since it involves manually editing a critical data file.

The MAZE package significantly improves this process by introducing the make method. The make method takes a zeolite IZA code as input, looks for the corresponding CIF file, and if it is not found attempts to download the zeolite CIF file from the IZA database. After locating or downloading the correct CIF file, the make function then builds a Zeolite from a IZA CIF file, and stores the mapping between the indices and their identities in two of the Zeolite objects internal dictionaries. The identities of the sites can then be determined by using the get_site_type method or by accessing the dictionaries directly (Fig. 4).

4.2. Structural manipulations

The Atoms class' structural manipulation features allow atoms to be added and removed from the collection and the properties of individual atoms to be altered. The API by which these manipulations are performed is inspired by Python's list manipulation methods. Although familiar to Python users, these manipulations

```python
from maze import Zeolite
def change_atoms(atoms):
    for atom in atoms:
        if atom.symbol == 'Si' and atom.tag != 154:
            atoms[atom.index].symbol = 'Sn'

bea = Zeolite.make('BEA')
cluster = bea.get_cluster(154)[0].cap_atoms().apply(change_atoms).remove_caps()
bea_sn = bea.integrate(cluster)
```

Fig. 6. Code required to generate structure F from structure A.

are not self-consistent as some have side effects (e.g., the pop method) while others are side effect free such as the __add__ method. In zeolite workflows, it is common for many derivatives of a single parent zeolite to be generated, and this is complicated by methods with side-effects, due to the need for explicit copying prior to each modification.

In alignment with the goal of the MAZE project, new methods for atomic manipulation were designed, which do not mutate the underlying object, and instead return a copy with the applied modifications. These methods (add_atoms and delete_atoms) simplify the computational workflows and also allow for method chaining improving code readability. A list of the available methods for the ASE Atoms object and the MAZE Zeolite object are shown in table S1.

4.3. Cluster extraction, atom capping and integration

The power of these additional structural manipulation features can be demonstrated by performing a complex workflow. A typical zeolite unit cell contains over one-hundred atoms, but the region of chemical interest is frequently confined to the atoms adjacent to a few T-sites. To reduce the computational expense of quantum chemical calculations, the calculations are typically performed on a smaller subset of atoms adjacent to the active sites of interest. This subset of atoms is referred to as a cluster [18]. Capping atoms (usually, hydrogens) are added to the terminal cluster atoms to make chemically meaningful structures. The optimal position for the capping atoms is based on the parent zeotype's structure. After the capped cluster's structure has been optimized, the cluster can be integrated back into the initial zeolite for further downstream analysis.

This workflow is extraordinarily difficult to perform with the ASE base package due to the challenge associated with tracking the relationship between atom indices during the extraction, manipulation, and reinsertion step. The Zeolite class's built-in index mapper ensures that the relationship between atoms can easily be determined and forms the basis for the simple functions that perform this workflow. In Fig. 5 a pictorial representation of stages in the workflow is shown along with the methods needed to perform the transformation from one stage to the next.

The overall workflow has six distinct structures bridged by functions which take the previous structure as an input and output the new structure. The cluster structures (B, C, D, E) have different indices than the BEA frameworks (A, F), yet the indices can easily be mapped to each other using the built-in IndexMapper's get_index method. Since the functions do not alter the zeolite to which they are applied, and instead return a new zeolite object, they can be chained together. The chained methods required to transform structure A into structure F is shown in Fig. 6.

The code presented in Fig. 6 demonstrates how a complex workflow can be achieved with the chaining of several functions together. This simplicity allows for knowability of the operations,

precise and complete operability, and robustness due to high readability. The scrambling of the indices with the cluster extraction does not allow for consistent code using the base ASE package. Instead, the indices of each atom must be matched manually at each stage of the process. Thus, the MAZE package interface has increased the knowability, operability and robustness compared to the cumbersome manual workflow required when using the base ASE package.

5. Impact and conclusion

The improved API of the MAZE package was presented here by demonstrating how to perform representative tasks. Several other features of the MAZE package include database integration and adsorbate additions. A complete description can be found in the documentation, which is referenced in the supplementary material. MAZE's improved API builds on-top of the Atomic Simulation Environment. This new interface facilitates computational zeolite calculations by greatly simplifying the steps needed to perform common zeolite tasks. [2] Computational experiments are less labor intensive than wet lab experiments, but lack of optimal APIs for scientific software and complex workflows can incur a significant time commitment from researchers to setup and run. By creating custom software tailored to the specific task, research can be simplified and larger scale experiments can be conducted.

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.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.softx.2021.100797.

References

[1] Smit B, Maesen TLM. Towards a molecular understanding of shape selectivity. Nature 2008;451(7179):671–8. http://dx.doi.org/10.1038/nature06552.

[2] Xie P, et al. Bridging adsorption analytics and catalytic kinetics for metal-exchanged zeolites. Nat Catal 2021;4(2):144–56. http://dx.doi.org/10.1038/s41929-020-00555-0.

[3] Kosinov N, Liu C, Hensen EJM, Pidko EA. Engineering of transition metal catalysts confined in zeolites. Chem Mater 2018;30(10):3177–98. http://dx.doi.org/10.1021/acs.chemmater.8b01311.

[4] Mohan S, Dinesha P, Kumar S. NOx reduction behaviour in copper zeolite catalysts for ammonia SCR systems: A review. Chem Eng J 2020;384:123253. http://dx.doi.org/10.1016/j.cej.2019.123253.

[5] Sushkevich VL, van Bokhoven JA. Methane-to-methanol: Activity descriptors in copper-exchanged zeolites for the rational design of materials. ACS Catal 2019;9(7):6293–304. http://dx.doi.org/10.1021/acscatal.9b01534.

[6] Astala R, Auerbach SM, Monson PA. Density functional theory study of silica zeolite structures: Stabilities and mechanical properties of SOD, LTA, CHA, MOR, and MFI. J Phys Chem B 2004;108(26):9208–15. http://dx.doi.org/10.1021/jp0493733.

[7] Zheng X, Blowers P. A computational study of methane catalytic reactions on zeolites. J Mol Catal A: Chemical 2006;246(1–2):1–10. http://dx.doi.org/10.1016/j.molcata.2005.10.009.

[8] Catlow CRA, Van Speybroeck V, van Santen R. Modelling and Simulation in the Science of Micro- and Meso-Porous Materials. Saint Louis, UNITED STATES; 2017, [Online]. Available http://ebookcentral.proquest.com/lib/ucdavis/detail.action?docID=5051428.

[9] Weckhuysen BM, Yu J. Recent advances in zeolite chemistry and catalysis. Chem Soc Rev 2015;44(20):7022–4. http://dx.doi.org/10.1039/C5CS90100F.

[10] Himanen L, Geurts A, Foster AS, Rinke P. Data-driven materials science: Status, challenges, and perspectives. Adv Sci 2019;6(21):1900808. http://dx.doi.org/10.1002/advs.201900808.

[11] Pirhadi S, Sunseri J, Koes DR. Open source molecular modeling. J Mol Graph Model 2016;69:127–43. http://dx.doi.org/10.1016/j.jmgm.2016.07.008.

[12] Kresse G, Joubert D. From ultrasoft pseudopotentials to the projector augmented-wave method. Phys Rev B 1999;59(3):1758–75. http://dx.doi.org/10.1103/PhysRevB.59.1758.

[13] Plimpton S. Fast parallel algorithms for short-range molecular dynamics. J Comput Phys 1995;117(1):1–19. http://dx.doi.org/10.1006/jcph.1995.1039.

[14] Mortensen JJ, Hansen LB, Jacobsen KW. Real-space grid implementation of the projector augmented wave method. Phys Rev B 2005;71(3):035109. http://dx.doi.org/10.1103/PhysRevB.71.035109.

[15] Hjorth Larsen A, et al. The atomic simulation environment—a Python library for working with atoms. J Phys: Condens Matter 2017;29(27):273002. http://dx.doi.org/10.1088/1361-648X/aa680e.

[16] Mosqueira-Rey E, Alonso-Ríos D, Moret-Bonillo V, Fernández-Varela I, Álvarez-Estévez D. A systematic approach to API usability: Taxonomy-derived criteria and a case study. Inf Softw Technol 2018;97:46–63. http://dx.doi.org/10.1016/j.infsof.2017.12.010.

[17] Bahn SR, Jacobsen KW. An object-oriented scripting interface to a legacy electronic structure code. Comput Sci Eng 2002;4(3):56–66. http://dx.doi.org/10.1109/5992.998641.

[18] Schroeder C, et al. A stable silanol triad in the zeolite catalyst SSZ-70. Angew Chem Int Ed 2020;59(27):10939–43. http://dx.doi.org/10.1002/anie.202001364.