The Journal of Chemical Physics

stk: An extendable Python framework for automated molecular and supramolecular structure assembly and discovery

Cite as: J. Chem. Phys. **154**, 214102 (2021); https://doi.org/10.1063/5.0049708
Submitted: 08 March 2021 . Accepted: 12 May 2021 . Published Online: 01 June 2021

Lukas Turcani, Andrew Tarzia, Filip T. Szczypiński, and Kim E. Jelfs

COLLECTIONS

Paper published as part of the special topic on Computational Materials Discovery

View Online Export Citation CrossMark

ARTICLES YOU MAY BE INTERESTED IN

Machine learning meets chemical physics
The Journal of Chemical Physics **154**, 160401 (2021); https://doi.org/10.1063/5.0051418

r^{2}SCAN-3c: A "Swiss army knife" composite electronic-structure method
The Journal of Chemical Physics **154**, 064103 (2021); https://doi.org/10.1063/5.0040021

Machine learning of free energies in chemical compound space using ensemble representations: Reaching experimental uncertainty for solvation
The Journal of Chemical Physics **154**, 134113 (2021); https://doi.org/10.1063/5.0041548

© 2021 Author(s).

The Journal of Chemical Physics ARTICLE scitation.org/journal/jcp

stk: An extendable Python framework for automated molecular and supramolecular structure assembly and discovery

Cite as: J. Chem. Phys. 154, 214102 (2021); doi: 10.1063/5.0049708
Submitted: 8 March 2021 • Accepted: 12 May 2021 •
Published Online: 1 June 2021

View Online Export Citation CrossMark

Lukas Turcani, Andrew Tarzia, Filip T. Szczypiński, and Kim E. Jelfs[a)]

AFFILIATIONS

Department of Chemistry, Molecular Sciences Research Hub, Imperial College London, White City Campus, Wood Lane, London W12 0BZ, United Kingdom

Note: This paper is part of the JCP Special Topic on Computational Materials Discovery.
[a)]**Author to whom correspondence should be addressed:** k.jelfs@imperial.ac.uk

ABSTRACT

Computational software workflows are emerging as all-in-one solutions to speed up the discovery of new materials. Many computational approaches require the generation of realistic structural models for property prediction and candidate screening. However, molecular and supramolecular materials represent classes of materials with many potential applications for which there is no go-to database of existing structures or general protocol for generating structures. Here, we report a new version of the supramolecular toolkit, *stk*, an open-source, extendable, and modular Python framework for general structure generation of (supra)molecular structures. Our construction approach works on arbitrary building blocks and topologies and minimizes the input required from the user, making *stk* user-friendly and applicable to many material classes. This version of *stk* includes metal-containing structures and rotaxanes as well as general implementation and interface improvements. Additionally, this version includes built-in tools for exploring chemical space with an evolutionary algorithm and tools for database generation and visualization. The latest version of *stk* is freely available at github.com/lukasturcani/stk.

© 2021 Author(s). All article content, except where otherwise noted, is licensed under a Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/). https://doi.org/10.1063/5.0049708

I. INTRODUCTION

Computational modeling seeks to accelerate functional material discovery and support experimental workflows by offering insights into chemical processes and structures that are not achievable through experiment. With advances in hardware and software, it is now possible to couple computational and experimental exploration of the vast array of potential materials and their properties at a much lower time and resource cost than experiment alone and with a lower risk of wasted effort.[1–3] Artificial intelligence (AI) and machine learning have the potential to assist in the efficient exploration of known and unexplored chemical space toward the optimal materials for a specific application.[4] For example, AI-driven computational workflows have been applied to explore the chemical space of transition metal complexes[5–7] and organic electronics.[8,9] Many of these approaches are facilitated by a strong push in the materials modeling community to develop open-source repositories of code, material structures, and their properties.

There are established computational methods (of varying cost and accuracy) for calculating the properties of materials for many of the problems in materials science. Such approaches have facilitated the prediction of the properties of hypothetical and known materials for screening toward particular applications. However, the accurate calculation of many material properties requires a realistic and representative structural model. One solution to this problem is to use an existing database of structures from experimental results and screen them for their properties, which is a common approach in solid-state materials or biological materials.[10–13] Unfortunately, such an approach limits exploration beyond known examples. Therefore, the ideal solution for novel materials discovery is to generate structures from scratch. Research groups often employ

© Author(s) 2021

in-house scripts written for specific material types to generate structures for high-throughput materials screening. As such, scripts are often tailor-made for the groups' specific needs and are not made available to the broader scientific community. Furthermore, they are hard to maintain and difficult to generalize to a broader system set. In particular, the structure prediction of organic and supramolecular materials is currently difficult to generalize and limited to a small subset of possible chemical classes. There are programs currently available (some of which are open-source) for the generation of materials such as metal–organic polyhedra,[14,15] organic and inorganic molecules,[16,17] and polymeric systems.[18–26] However, it is not trivial to interface such codes for more general workflows or more complex projects and the structures that are constructable by these software are specifically focused.

To tackle the problem of general structure generation for materials discovery, we have previously reported developing the supramolecular toolkit (*stk*): an open-source and easily extendable Python library for the assembly of complex molecular architectures, including supramolecular assemblies.[27] Importantly, *stk* mainly generates molecular representations of materials and the problem of crystal structure prediction, or how molecules pack in the solid-state, is beyond the scope of this work. Here, we provide an update on the *stk* structure generation software, an extension to new material classes, and advanced capabilities including chemical space exploration. We have rewritten *stk* to focus on constructing a diverse range of structures with an easy-to-use interface and modular, modifiable functionality. Our code development ideology in *stk* provides user-friendly defaults for generating molecules based on an underlying topology while allowing for the safe extension to new approaches (e.g., structure construction without an underlying topology) or materials classes without requiring a brand-new implementation. The modular design of *stk* makes the addition of new structure classes and reactions as simple as possible for end users. In addition, *stk* allows users to deposit molecules, and molecular properties, easily into local or remote MongoDB databases. MongoDB databases constructed by *stk* can be viewed through *stk-vis*,[28] a stand-alone, cross-platform application providing 2D and 3D molecular rendering and molecular property tabulation. Molecules deposited by *stk* into MongoDB databases are immediately visible in *stk-vis*, allowing users to easily share computationally constructed molecular databases both within and across teams.

On top of the previously reported construction of covalent systems, such as linear polymers, covalent organic frameworks (COFs), and porous organic cages,[27] the most recent version of *stk* allows for the inclusion of metal centers, thus opening the scope to a diverse range of metal–organic cages and metal–organic frameworks (MOFs). Furthermore, *stk* now allows for the automated and custom construction of rotaxanes alongside existing supramolecular structures, such as host–guest complexes. Figure 1 shows examples of *stk* constructed molecules from each broad topology type already implemented in *stk*. Finally, *stk* provides convenient methods for interfacing with third-party software for geometry optimizations and property calculations of *stk*-generated molecules. Geometry optimization is beyond the scope of *stk*; for this, we have written the open-source repository *stko* that contains functions for the geometry optimization and analysis of molecules (github.com/JelfsMaterialsGroup/stko).[29]

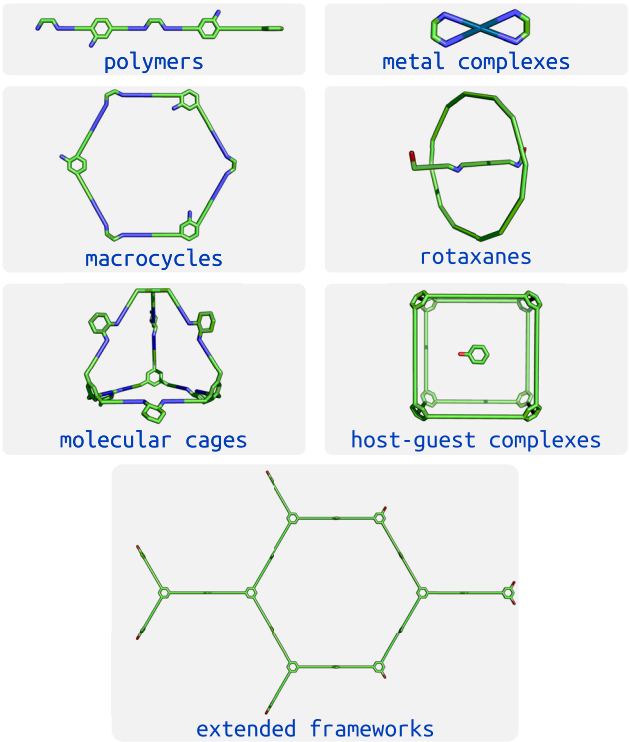


FIG. 1. Examples of buildable topology types with *stk*. Names do not match their name in the *stk* code. These structures represent the placement and alignment of building blocks on a topology graph and do not have chemically realistic bonds lengths and angles between building blocks as they are not geometry optimized here.

This paper describes the construction of numerous molecule types and their associated construction approaches currently implemented in *stk*. Additionally, we describe the user interfaces provided by *stk* for making databases of molecules and the use of an evolutionary algorithm (EA) to explore chemical space with *stk*. Further examples and more thorough documentation can be found at https://stk.readthedocs.io; all *stk* codes are freely available at github.com/lukasturcani/stk, while *stk-vis* can be found at github.com/lukasturcani/stk-vis.

II. SOFTWARE OVERVIEW

Here, we describe the individual software components of *stk* building up to the use of the evolutionary algorithm. *stk* is a Python library that provides users with the following capabilities: (i) automated construction of diverse molecular and supramolecular models, regardless of their complexity; (ii) automatic design of molecules with user-desired properties; and (iii) creation of molecular databases. The default construction algorithm of *stk* is split into distinct algorithms that perform specific tasks (e.g., placement or reactions). These specific algorithms are fundamentally similar to other topology-based algorithms available in other software. However, *stk* provides a software library for implementing these techniques toward the construction of any desired molecule type. The

© Author(s) 2021

development of *stk* focuses on a robust and straightforward user interface, with the implementation details being ultimately hidden from the user. Therefore, the above capabilities of *stk* are easily accessible with limited programming experience. For a tutorial-style introduction to using *stk*, we recommend visiting the documentation at https://stk.readthedocs.io. Additionally, while *stk* provides built-in examples for each of its features, a primary design goal is that users may extend any aspect of *stk* in their code, without touching the source-code of *stk* itself. Throughout this paper, we describe the default implementation of *stk*.

The interface of *stk* handles the input, construction, and output of molecules (Fig. 2). While previously *stk* provided an interface for interacting with third-party optimization software, this functionality has since been removed, as there already exists a well-developed Python ecosystem for providing this functionality.[30–32] This reduction in scope allows the development of *stk* to focus on its key features. However, many of the previously implemented protocols (including Schrödinger's Macromodel,[33] GULP,[34,35] xTB,[36] and RDKit;[37] these software packages provide access to geometry optimizations, property calculations, dynamics simulations, and conformer generation algorithms) are available in our repository *stko*.[29] Recently, we added open-source optimization protocols to the construction process; these protocols were added into *stk* because they do not introduce significant software dependencies. Additionally, we have developed *stk-vis*,[28] which is a cross-platform application for the visualization of databases created by *stk* (Sec. II F). *stk-vis* can also connect to remote MongoDB databases of *stk* molecules, facilitating sharing among researchers.

A. Construction overview

The primary process that *stk* performs, which facilitates most of its capability, is the construction of constructed molecules from a topology graph and building blocks (represented by the **ConstructedMolecule**, **TopologyGraph**, and **BuildingBlock** classes, respectively; the bold text represents a class name within *stk*). The default implementation of the construction process occurs in stages: (1) **BuildingBlock** instances are placed and aligned on vertices of a **TopologyGraph**; (2) **FunctionalGroup** instances of those **BuildingBlock**s are assigned to the edges of the **TopologyGraph**; (3) "reactions" are performed to connect functional groups assigned to the same edge; and (4) (as an optional final step) the geometry of the structure is optimized. For example, Fig. 3 shows the placement of building blocks [separated molecules in (b)] on a topology graph to form a constructed molecule. The topology graph used to construct a molecule defines the underlying algorithms of each step, which are described in short below. Section III shows that *stk* allows the construction of broad classes of molecules using this simple default implementation, where distinctions between molecules come from the specific algorithms implemented in their topology graph.

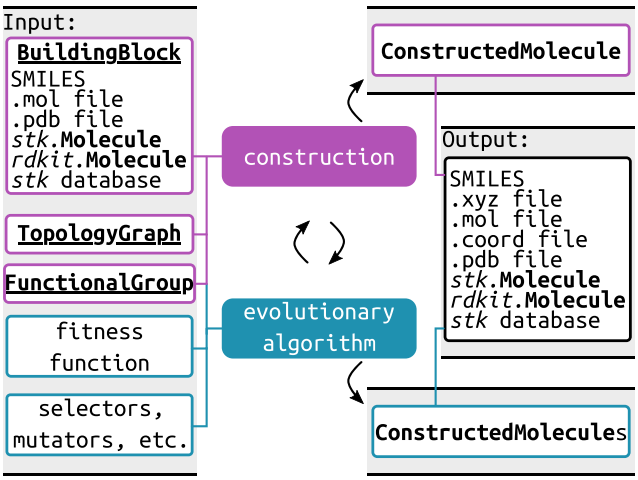


FIG. 2. The connection between the user-input, *stk* construction and evolutionary algorithms, and output. The complexity of *stk* is behind the construction and evolutionary algorithms, while the user interface is as simple as possible. We highlight the variety of input and output options, which allow for interfacing with other computational chemistry software. The construction algorithm is the crucial component behind *stk* usage.

B. Topology graphs, vertices, and edges

In the default implementation of *stk*, molecules are constructed by placing building blocks on topology graphs [Figs. 3(a) and 3(b)]. Topology graphs construct molecules by first defining an underlying graph of vertices (**Vertex** instances) and edges (**Edge** instances) [Fig. 3(a)]. Each vertex in a topology graph defines a position, where the building block is placed (this can be defined in Cartesian coordinates or defined relative to the vertex's neighbors), and the series of transformations applied to the building block to align its functional groups with the neighboring edges. Edges in a topology graph define which functional groups are joined during the reaction step of the construction process. Additionally, the position of edges determines the orientation of the building blocks, where the transformation defined by the vertex aligns functional groups with their assigned edges. The implemented alignment algorithm of a vertex class, which is made up of a series of independent transformations, aims to orient a building block while minimizing distortion and clashes in the bonds generated during construction (Fig. 4).

Throughout *stk*, we have implemented robust alignment processes for the built-in topology graphs that vary in complexity. In Sec. III, we describe the built-in molecule types, all of which use similar but distinct topology graph, vertex, and edge classes that

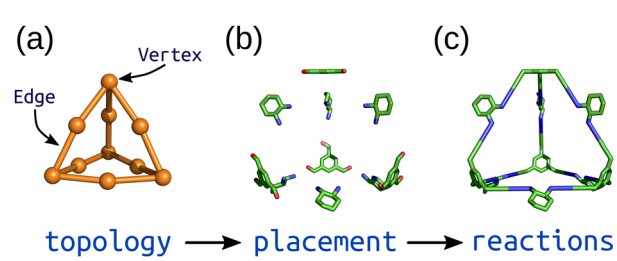


FIG. 3. Schematic of the construction process of an organic cage in *stk* starting from a (a) topology graph of vertices connected by edges. The supplied building blocks are (b) placed and aligned on the topology, and then (c) "reactions" are performed between them.

© Author(s) 2021

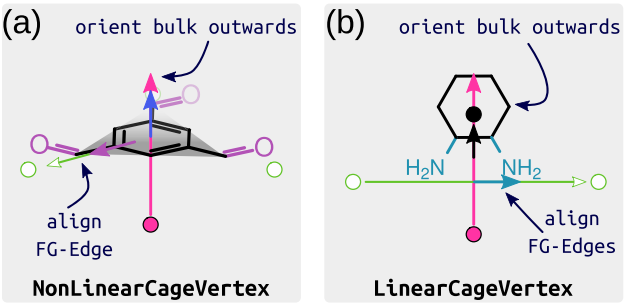


FIG. 4. Schematic of the alignment of a (a) **NonLinearCageVertex** and (b) **LinearCageVertex**, which are vertex classes specific to cage construction. These vertices use different approaches to orient the bulk (part of the chemical structure in black) of their building blocks away from the topology center (pink circle). Nonlinear building blocks align the normal (blue arrow) to the plane defined by its functional groups (shaded triangle) with the pink arrow, while the linear building blocks align the vector between the placer and core centroids (black arrow) with the pink arrow. Both instances use the position of the edge and functional group centroids to define the alignment of a functional group [part of the chemical structure in purple in (a) and cyan in (b)] with an edge (green circles).

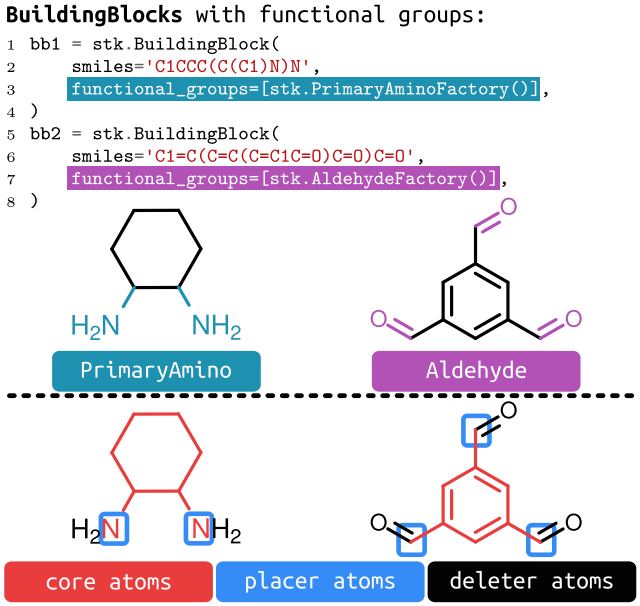


FIG. 5. Code snippet showing the generation of **BuildingBlock** instances of two molecules from their SMILES strings using built-in functional group factories. Functional group factories search the molecules for the user-requested functional groups as an alternative to having the user specify each functional group individually. Blue and purple colored boxes highlight the code used to request specific functional groups, shown in the same color on the middle chemical structures. Below the dashed lines, we show the four subsets of atoms defined upon building block initialization: *bonder* and *placer* (highlighted by blue outlines), *core* (colored red), and *deleter* atoms (colored black). By default, *placer* and *bonder* atoms are equivalent, which are automatically handled based on the functional groups present. However, they can also be user-defined.

are appropriate for their construction. Different chemical systems require different vertex transformations to achieve alignment successfully; for example, the processes that orient building blocks on a cage topology graph will differ from those that orient a cycle and axle to form a rotaxane molecule. However, the independent steps underlying these algorithms are simple and conserved among all vertex classes (i.e., they use the same code to transform the building blocks). Therefore, we have made the implementation of new topology graph, vertex, and edge classes straightforward such that the extension of *stk* to a user's materials is possible. Importantly, user-defined topology graph, vertex, and edge classes can use an approach that is equivalent to one of the built-in classes or an entirely new approach; in other words, the processes used by developers of *stk* are entirely customizable.

C. Building blocks and functional groups

The **BuildingBlock** class in *stk* represents a molecule, which is placed and aligned on the vertices of a topology graph. Building blocks will also be joined to other building blocks during the construction process, assuming an edge connects the vertices on which the building blocks are placed. The building block representation includes the atoms and bonds of a molecule, and its position matrix, which is a matrix of atomic coordinates. For placement, alignment, and reactions to be carried out, building blocks also contain **FunctionalGroup** instances, which are defined by the user using the **FunctionalGroupFactory** interface (Fig. 5).

A functional group of a building block defines three sets of atoms designated for use by the default construction process: *bonders*, *deleters*, and *placers* (Fig. 5). *Bonder* and *deleter* atoms represent the atoms that will bond and be deleted, respectively, during a reaction. *Placer* atoms are used to place and align the building block. Finally, a building block also has *core* atoms, representing the bulk of the molecule, which often needs to be oriented in a specific direction. Note that *core* atoms will not get modified by reactions during the construction process. All building blocks have defined *core* and *placer* atom sets, while a building block can have no functional groups and, as a result, no *bonder* or *deleter* atoms. Therefore, to define *bonder* and *deleter* atoms, a functional group must be defined.

Functional groups in *stk* are defined by searching the building block for the chemical pattern that represents the desired functional group. We provide many built-in functional groups in *stk* that cover common functionalities (such as alcohols, amines, aldehydes, and halogens), and the documentation covers the definition of new functional groups. Additionally, new chemical patterns can be defined very simply using SMARTS strings (a string-based representation of chemical patterns) to search for functional groups within the molecule using RDKit.[37] Therefore, *stk* can handle arbitrary chemical transformations of interest to the user. Importantly, this approach to defining functional groups in *stk*, through the **FunctionalGroup** classes, is as user-friendly and straightforward as possible.

D. Reactions

Reactions are the algorithms that *stk* uses to connect building blocks during the construction process. We must emphasize that

© Author(s) 2021

we discuss reactions on topology graphs from the perspective of the implementation within *stk*, not the associated chemical process. **Reaction** classes define algorithms that act on functional groups to either add or remove atoms or bonds. The addition of bonds between atoms of functional groups of different building blocks is what ultimately leads to building blocks being joined by the construction process. Steps two and three [Figs. 3(b) and 3(c)] of the default construction process define, then perform, reactions between building blocks. In the case of vertices connected by edges, the second step assigns each functional group on a **BuildingBlock** to an edge connected to that vertex. In step three, each edge is designated a reaction based on the set of associated functional groups, and then *stk* performs those reactions. *stk* provides three generic reactions that can be used with a wide array of functional groups: **OneOneReaction**, **OneTwoReaction**, and **TwoTwoReaction**, which work on pairs of functional groups with the following combinations of *bonder* atoms: one and one, one and two, and two and two. In these cases, the algorithm is simple but generally applicable: a generic reaction between two such functional groups will delete *deleter* atoms and form bonds between *bonder* atoms on different functional groups. For a **TwoTwoReaction**, the ambiguity between which *bonders* get connected is resolved by bonding the two closest atoms and then the second nearest set.

Reactions also have the capability to create bonds with any bond order desired by the user and dative bonds for use with metal-coordinating species. As with the other parts of *stk*, the reaction process has robust defaults, but these are very customizable if need be; that is, the user can override which reactions are used to react specific pairs of functional groups when creating a **ConstructedMolecule**. By default, *stk* will select the reaction (from **OneOneReaction**, **OneTwoReaction**, and **TwoTwoReaction**) that matches the number of *bonder* atoms in the functional groups. Our interface for chemical reactivity in *stk* focuses on simplicity and generalizability. Importantly, arbitrary functional groups and reactions can be defined using the provided interface, which affords the general applicability of *stk* to user-defined problems. The default options for reactions (provided in the online documentation) are well suited to most uses of *stk*.

E. Modular and independent construction steps

Here, we describe the default implementation of molecule construction in *stk*, which is performed at the level of vertices and edges; this means each vertex performs an independent, self-contained, operation on a single building block. Similarly, reactions between functional groups on one edge are entirely independent of those on other edges. Therefore, the construction process is local to a single building block or a pair of building blocks for the reaction step and is trivially parallelizable. Additionally, by separating the entire process into smaller, well-defined algorithms, the code is easier to test, extend, and maintain. A topology graph can define arbitrary vertex and edge geometries, making the structure space accessible by *stk* infinitely extendable. For example, vertices do not need to be connected by edges, which results in nonreactive topology graphs that are crucial for the study of supramolecular materials (Sec. III D).

The main focus of *stk* is implementing robust and general construction algorithms using idealized topologies. However, this process results in a nonphysical structure where the connectivity between building blocks is exaggerated in distance. We recognize the need for chemically reasonable structures and have implemented an interface for third-party software in our repository *stko*[29] and recently added two open-source geometry optimization processes to the construction process within *stk* that are accessible as optional arguments. The newly implemented geometry optimization protocols (part of the MCHammer package that is available at github.com/andrewtarzia/MCHammer) work to decrease the distance between building blocks on a topology graph by performing rigid translations of the building blocks after their reaction. These processes are nonphysical and, as a result, are generally applicable to any **ConstructedMolecule**. However, their lack of physical meaning suggests that they should be used with care and perhaps as the initial step in further optimization sequences.

F. Databases

A significant aspect of computational structure generation is developing and sharing databases of structures and their properties. As such, *stk* provides built-in support for depositing molecules and their properties into MongoDB databases; the database interface is generalizable to other database schemas. *stk* supports three database types: **MoleculeDatabase**, **ConstructedMoleculeDatabase**, and **ValueDatabase**. **MoleculeDatabase** and **ConstructedMoleculeDatabase** databases are used for storing molecules (atoms, bonds, and position matrices) and **ValueDatabase** is used for storing properties in the form of strings, numbers, lists, dictionaries, and nested dictionaries thereof. Notably, the constructed molecules maintain the information about the building blocks used to construct the molecule, which are also stored in the database; these building blocks do not maintain the functional groups used.

Here, we introduce a secondary piece of software, *stk-vis*,[28] an open-source, cross-platform application for browsing local and remote *stk*-generated MongoDB databases. For each entry in the database, *stk-vis* provides visualization of the properties (from a **ValueDatabase**) and 2D representation and 3D structure (from the stored position matrix). Additionally, if visualizing constructed molecules, *stk-vis* makes the inspection of its constituent building blocks very simple. *stk-vis* facilitates the tabulation of molecules and their properties, including sorting based on a specific property and sharing this interface through a single file transfer. Figure 6 shows an example of the *stk-vis* graphical interface.

G. Evolutionary algorithm

Evolutionary algorithms (EAs) are efficient approaches for exploring chemical space.[38–41] EAs mimic the evolutionary process by taking some population, performing mutation and crossover events on that population, and then selecting survivors for the next generation. They are highly flexible algorithms, where the definition of the genome of the population and the way the population is selected, modified, and ranked are all modifiable by the user. This flexibility allows EAs to be applied in a variety of fields. The fragment-based approach treats members of the population as a collection of components that can all be modified. Such an approach is

© Author(s) 2021

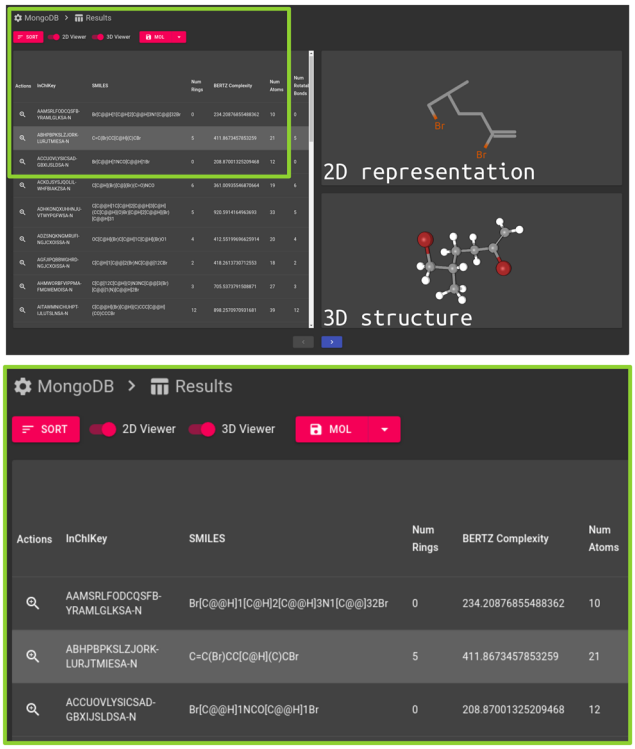


FIG. 6. Screenshot of the *stk-vis* graphical interface showing a table of molecule properties in a database and the 2D and 3D representations of the selected molecule. A zoom-in on part of the database table bounded by the green box is shown at the bottom.

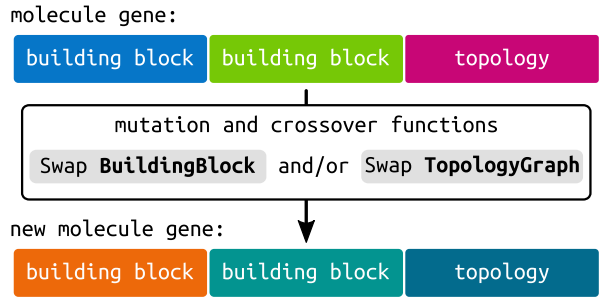


FIG. 7. Schematic of the modification of the generalized molecule gene available within *stk*, where a molecule gene is defined by the constituent building blocks and underlying topology graph, to a new molecule gene based on mutation or crossover functions. Here, colors represent a change in the building block or topology graph.

amenable to the construction approach of *stk*, where molecules are constructed from a combination of building blocks and a topology graph. We provide a general and modular implementation of the components of an EA in *stk* to automate materials discovery using the fragment-based approach.[42,43]

Given a population of molecules, *stk*'s EA provides a series of functions for selection, mutation, crossover, and fitness function calculation and normalization. The EA works specifically on the **ConstructedMolecule** class and uses, by default, the building blocks and topology graph as the gene (Fig. 7). Therefore, *stk* explores mutations of that gene toward constructed molecules with the desired properties by modifying the constituent building blocks or topology graph; this process uses the *stk* construction process to generate new candidates. Finally, the EA in *stk* directly feeds its results into *stk* databases and *stk-vis* for real-time collaboration.

The entire EA process can be user-defined for a specific problem, i.e., automating the search for molecules with user-defined properties. Fitness functions, in particular, must be provided to the EA and are regular Python functions that take a **ConstructedMolecule** and return a value representing its fitness. *stk* provides multiple selection, mutation, and crossover algorithms for an EA. All implemented algorithms use a fragment-based approach, where each building block is treated as a fragment of its corresponding constructed molecule. Therefore, the implemented mutation and crossover algorithms work at the building block-level by mutating or swapping building blocks in constructed molecules. For example, the **RandomBuildingBlock** mutation will switch out the building block used in construction with a random replacement from some population of building blocks and construct a new molecule. Additionally, the user can mutate the topology graph of a constructed molecule to explore entirely distinct structures. *stk* provides a robust interface for implementing arbitrary fragment-based EAs and is continuously in development. Importantly, the documentation provides thorough examples of implementing the EA from scratch.

III. EXAMPLES OF IMPLEMENTED SYSTEMS

In Secs. III A–III E, we highlight the materials classes that *stk* can construct, including molecular materials and extended framework materials. Molecular materials are discrete and include examples of varying complexity, such as linear polymers, macrocycles, metallocycles, organic and metal–organic cages, catenanes, rotaxanes, knots, and molecular machines. Extended materials are periodic and include metal–organic frameworks (MOFs) and covalent organic frameworks (COFs), which can be two- or three-dimensional. Importantly, *stk* constructs materials from building blocks and a topology graph and can assemble materials with covalent, coordination, or noncovalent interactions. Such an approach is effectively similar to the synthetic processes used for many supramolecular materials and other molecular materials, such as cage-like molecules and crystalline frameworks. Ultimately, *stk* allows for the structure generation of molecules of arbitrary complexity.

Given a topology graph and the appropriate functional groups, *stk* can, in principle, build any structure type seen in materials chemistry and, indeed, other fields of chemistry. Crucially, the construction interface in *stk* provides the necessary control over relative building block orientation and placement on a topology graph to allow for the construction of different structural isomers of a constructed molecule. The critical distinction between material types is that their underlying topology graph will define a specific series of

© Author(s) 2021

construction steps and vertices with specific alignment processes. Here, we describe the implemented topology graphs and corresponding molecule types, together with some examples of their use. In all cases, we show the structures directly output by *stk* without any geometry optimization. Importantly, to add new topologies, of molecule types similar to those below, only the definition of the new idealized geometry and its connectivity is required.

A. Polymers and macrocycles

Previously, we introduced the *stk* interface for constructing **Linear** polymers of any size with arbitrary repeat unit sequence and directionality.[27] We have added the **Macrocycle** topology graph, which allows for the construction of macrocyclic structures using a similar interface and process to the **Linear** polymer class. Both topology graphs take *repeating_unit* and *monomer_orientation* information as input to give the user full control over the order and orientation of building blocks in the polymer or macrocycle chain (i.e., to control configurational isomers). In both cases, the placer atoms in the building block's functional groups are used to align the building blocks [producing the black arrows in Fig. 8(a)]. This interface allows for the user to set fractional probabilities of building block "flipping" on the topology graph [purple arrows in Figs. 8(b) and 8(c)]. To date, the **Linear** polymer class has been used to explore very large (~200 000 molecules) chemical spaces of organic aromatic molecules.[9,44–49] Figures 8(b) and 8(c) show that with a family of building blocks at hand [Fig. 8(a)], we can easily construct arbitrary **Linear** polymers and **Macrocycle** structures.

B. Metal-complex construction

In this release of *stk*, we have added **MetalComplex** topology graphs, which handle the placement and alignment of metal atoms and ligands on metal-complex geometries. Other groups have recently implemented tools for constructing metal complexes and small molecules that encompass a broader set of possible metal-complex geometries than *stk*.[5,16,17] While *stk* can, in principle, be extended to construct any metal-complex geometry, we have focused on common geometries in supramolecular chemistry [e.g., variations of square planar and octahedral complexes, porphyrin, and paddlewheel geometries; Fig. 9(a)]. Other than the porphyrin topology, *stk* currently only handles mono- and bidentate coordination geometries. Figure 9(b) shows a code-snippet example of the definition of a palladium(II) atom and the subsequent assembly of a bidentate square planar complex with it.

MetalComplex graphs have specific metal-type and ligand-type vertices, where metal-vertices are single atoms and do not undergo any orientation. All ligand orientation is based on aligning ligand-binding sites (defined by **FunctionalGroup** instances) and the defined location of the edges in the **TopologyGraph**. Importantly, these topology graphs strictly define the position of **Edge** instances, which represent the ideal position of metal-coordinating atoms in the complex geometry. For bidentate ligands, the alignment process requires two idealized **Edge** positions to align the two functional groups on the ligand. This process appropriately enforces the alignment of the ligand bulk away from the metal-center and the two ligand-binding sites inline with the two metal-binding sites (defined by the **TopologyGraph**).

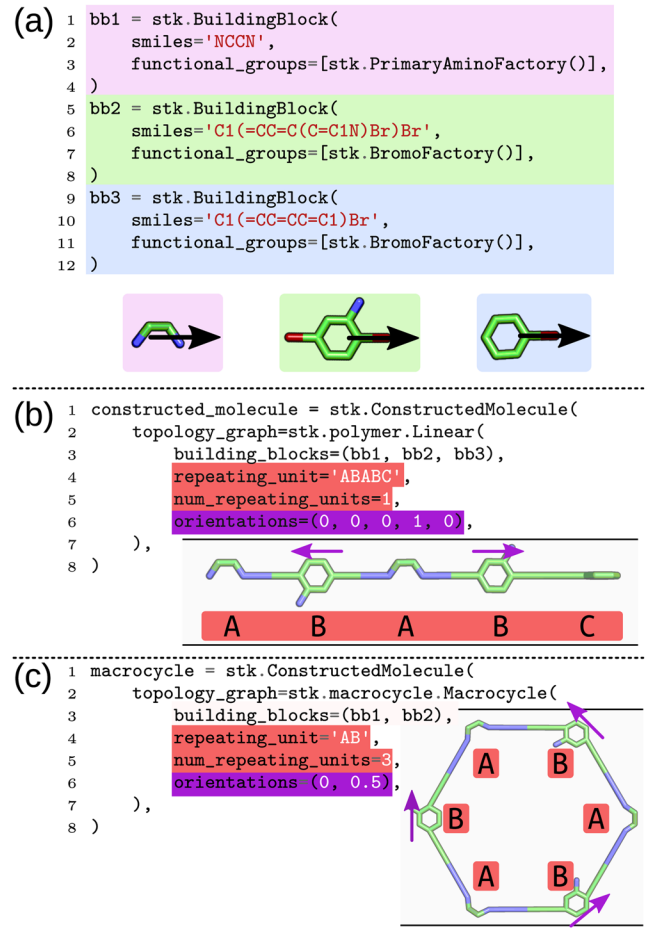


FIG. 8. (a) Definition of three building blocks in *stk*. The resulting molecules are visualized in the colored boxes with their orientation vectors (black arrows) highlighted. Assembly of (b) **Linear** and (c) **Macrocycle** topology from the defined building blocks in (a). Building block ordering and repeats are highlighted in red and orientations are highlighted using purple arrows (for building blocks that are not symmetric). Colored boxes match code snippets with their structure or effect on the structure.

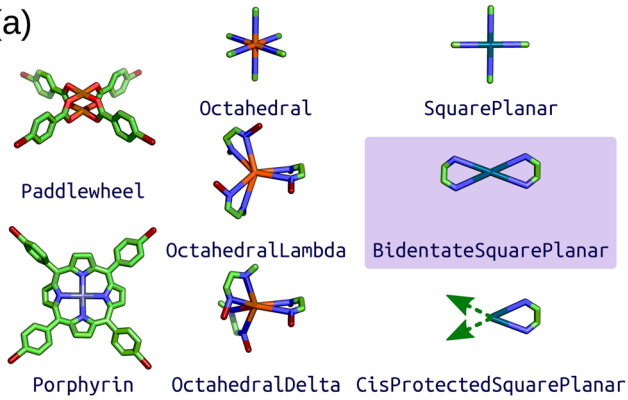
To handle metal-containing systems, *stk* allows for the definition of dative bonds. Examples of defining a **Reaction** that produces dative bonds are available in the documentation at https://stk.readthedocs.io. Finally, our approach to metal geometries requires the strict definition of a metal-complex geometry by the user (this has been completed for the implemented examples). Therefore, distinct **TopologyGraph** classes are required to handle the following use-cases (for example): (i) assembly of octahedral symmetries (Λ vs Δ symmetry of tris-bidentate octahedral complexes) and (ii) assembly of *cis*-protected square planar metal-complexes with free binding sites for further reaction [see Fig. 9(a)].

C. Molecular cages

Molecular cages are a broad class of molecular systems that may have an internal cavity (e.g., porous molecular cages). Cages are

© Author(s) 2021

(a)



(b)

```
pd_atom = stk.BuildingBlock(
    smiles='[Pd+2]',
    functional_groups=(
        stk.SingleAtom(stk.Pd(0, charge=2))
        for i in range(4)
    ),
    position_matrix=[[0, 0, 0]],
)

constructed_molecule = stk.ConstructedMolecule(
    stk.metal_complex.BidentateSquarePlanar(
        metals=pd_atom,
        ligands=bidentate_ligand,
    ),
)
```

FIG. 9. (a) Implemented **MetalComplex** topology graphs. Green dashed arrows show where subsequent coordination may be performed for the **CisProtectedSquarePlanar** graph. (b) Code snippet showing the definition of a palladium(II) atom with four **SingleAtom** functional groups and the subsequent assembly of a **BidentateSquarePlanar** metal complex. Colored boxes match up to code snippets with their structure. The code snippet in (b) is from an online example and is not complete because initialization of a precursor is required.

(a)

```
bb1 = stk.BuildingBlock('NCCN', [stk.PrimaryAminoFactory()])
bb2 = stk.BuildingBlock(
    smiles='O=Cc1cc(C=O)cc(C=O)c1',
    functional_groups=[stk.AldehydeFactory()],
)
cage = stk.ConstructedMolecule(
    topology_graph=stk.cage.FourPlusSix((bb1, bb2))
)
```

(b)

topologies by num. connections:

(2)	M3L3triangle, M4L4Square
(2, 3)	2plus3, 4plus6, 4plus62 6plus9, 8plus12, 20plus30, M4L6TetrahedronSpacer
(2, 4)	2plus4, 3plus6, 4plus8, 5plus10, 6plus12, 8plus16, 10plus20, M2L4Lantern, M3L6, M4L8, M6L12Cube, M12L24, M24L48
(2, 5)	12plus30
(3)	1plus1, 2plus2, 4plus4, M4L4Tetrahedron, M4L6Tetrahedron
(3, 4)	6plus8, M6L2L3Prism, M8L6Cube

FIG. 10. (a) Code snippet showing cage construction from two building blocks. (b) Listing of all cage topology graphs built into *stk* separated by the number of connections or functional groups required by building blocks in the topology graph. Each row contains the name of topology graphs in *stk*. The column on the left shows the number of connection points of building blocks in each topology graph, i.e., (2, 3) implies that the building blocks have either two or three connection points.

commonly synthesized from a bottom-up building block approach and can be formed from purely organic building blocks[50] or building blocks that contain metal complexes.[51,52] They are candidate materials for solid-state and solution-phase applications in, for example, storage, separations, and catalysis.[50,53] Of particular interest is the modularity of their design process, where specific structures or properties are targeted by choice of its constituent building blocks. Modular design processes based on constituent building blocks are well suited to *stk*. Previously, we reported functionality for constructing porous organic cages with various topology graphs,[27] which has since been used in the high-throughput screening of porous organic cages,[42,43,54] and the generation of a large (~60 000 structures) cage database for training machine learning models to predict their stability.[55] Figure 10(a) shows a code snippet of cage construction using *stk*, highlighting its simplicity.

In this release, we have implemented the handling of metal-based systems toward the construction of metal–organic cages. Overall, we have implemented 31 distinct cage topologies [Fig. 10(b)] that encompass structures with diverse connectivities commonly seen in the literature. We split the topology graphs based on organic and metal–organic categories to aid the user experience and maintain domain-specific nomenclature.[56] However, *stk* does not make any technical distinction between them (i.e., metal-containing building blocks can be placed on an organic cage topology). The critical modification required to handle metal-containing cages is that cage topologies can now handle vertices where building blocks cannot be aligned because they are a single atom (e.g., a metal atom with **SingleAtom** functional groups). Young and co-workers recently developed *cgbind*, which is open-source software for the construction of a handful of metal–organic cage topologies.[15,57] In comparison, *stk*'s implementation is more general, but constructed molecules require further optimization compared to those generated using *cgbind*. However, we aim to overcome this using simple optimization algorithms that result in cage structures with reasonable geometries.

Molecular cages are constructed from building blocks of different connectivity [Fig. 10(b)], including building blocks with one to five connection points. Therefore, their alignment with the topology graph's edges can be more complicated. In *stk*, we define alignment procedures based on the number of connections a building block will have in a given topology by defining the **Vertex** class (the construction approach is described in Sec. II E). For example, vertices with two connection points require building blocks with two functional groups and are aligned using the **LinearCageVertex** process. Similarly, vertices with three or more connections require building blocks with three or more functional groups and are aligned using the **NonLinearCageVertex** process. Additionally, we provide

© Author(s) 2021

the **UnaligningVertex** class for building blocks that do not require alignment. The **NonLinearCageVertex** placement process happens in two steps (Fig. 4): (i) orientation of the building block such that the vector normal to the plane of the placer atoms aligns with the normal of the plane of edge positions and (ii) orientation of the building block such that a specific structural isomer is constructed. The user selects a structural isomer by specifying which functional group of their building block aligns with which edge on the topology graph using an input argument to the **TopologyGraph** class (examples are given in the documentation at https://stk.readthedocs.io). After a building block is oriented on a vertex, functional groups are assigned to edges following a vertex-specific protocol. Once all building blocks are placed, and functional groups assigned to edges, reactions are performed between functional groups assigned to the same edge. So far, we have found that these processes are robust to any topology (metal–organic or organic) we implement and are sufficient for avoiding collisions of building blocks and inter-building block bonds. However, all the topology graphs we have implemented to date are concave geometries, where the bulk of the building blocks also points away from the structure's center.

D. Nonreactive topology graphs: Rotaxanes and host–guest complexes

Nonreactive topology graphs construct molecules focusing on the relative spatial arrangement of the building blocks, where no bonds are created between them. Two examples discussed here are [*n*]rotaxanes and host–guest complexes. Rotaxanes are molecules in which a ring-shaped macrocycle is threaded on an axle with stoppers at each end of the axle.[58] The bulky stoppers on the axle prevent the macrocycle from slipping and, hence, the rotaxanes are mechanically interlocked, and the building blocks cannot be separated despite not being covalently bonded. The *n* in [*n*]rotaxane corresponds to the number of interconnected building blocks, so a single macrocycle on one axle would be called a [2]rotaxane. Host–guest complexes are complexes formed between a guest molecule encapsulated in the cavity of a host molecule.[59]

Here, we describe the construction of two nonreactive topology classes: **NRotaxane** and **Complex**. This differs from the synthetic process for rotaxanes, where a chemical reaction is required to form the mechanical bond holding the macrocycle on the axle. Importantly, *stk* does not attempt to model the realistic reaction processes and should be used in an alchemical way that simplifies the construction process as much as possible. We have implemented the **NRotaxane** class, which takes an axle and any number of cycles and assembles a rotaxane. Figure 11(a) shows the formation of an **NRotaxane** from building blocks constructed using the **Linear** and **Macrocycle** classes. The macrocycles are evenly spaced along the axle, with full control over their orientation (with respect to the direction of the axle) and sequence along the axle, and are placed such that the normal of the plane of best fit of the macrocycle is parallel to the axle.

We have introduced a general code to construct host–guest complexes. The method shown here is entirely generalizable to any two *stk* **BuildingBlock** instances but focuses on the relative orientation and placement of a guest molecule to a host molecule [Fig. 11(b)]. To handle the guest's orientation relative to the host (e.g., to align a functional group with a specific binding site), the

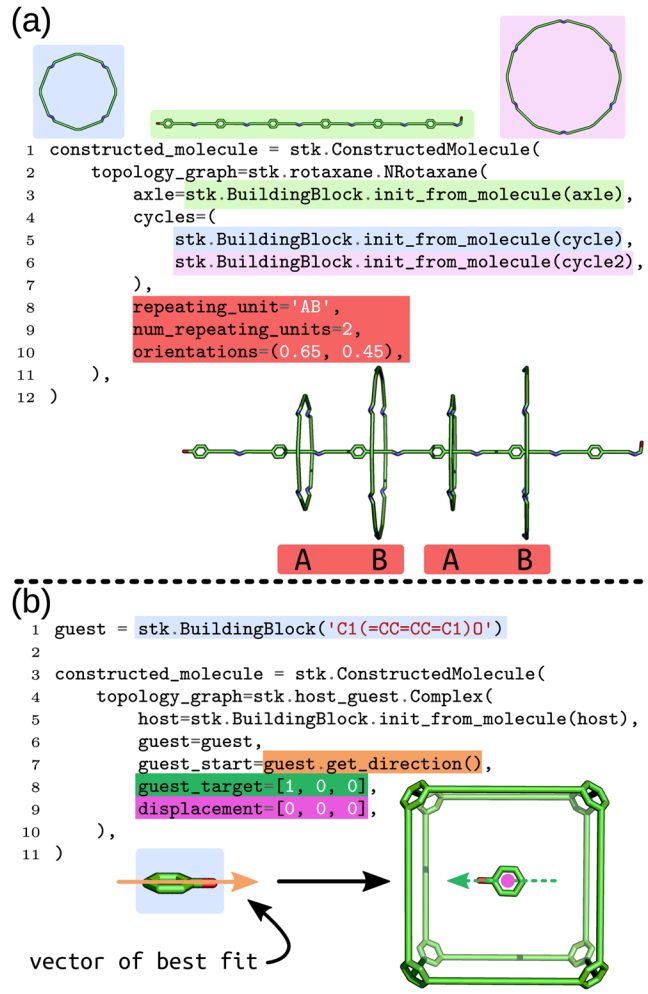


FIG. 11. (a) Assembly of an **NRotaxane** from an axle and two distinct macrocyclic building blocks. The code snippet highlights the possible modifications to the rotaxane structure by specifying the repeating unit, the number of repeating units, and orientations. (b) Assembly of a **Complex** from a host and guest. The code snippet highlights the methods and vectors used for guest orientation within the host structure. Colored boxes match up to code snippets with their structure, impact on a structure, vector, or position. Code snippets are not complete and require initialization of precursors.

Complex can be provided with initial and final vectors, where the guest is rotated such that the initial and final vectors are parallel. Figure 11(b) shows how to use the vector of best fit through a building block's atoms, to align the guest along a particular vector. By additionally defining the guest's translation relative to the host's centroid, the user can explore many host–guest conformations using *stk*. A recent example of host–guest structure generation in metal–organic cages[15] shows efficient ways of determining optimal guest orientation and placement, which can be automated with *stk* for any molecule class. Using low-cost simulation methods and the EA in *stk* (Sec. II G), we showed this on the simple case of C_{60} encapsulation in porous organic cages.[43] This work highlighted the

© Author(s) 2021

benefit of fragment-based host evolution toward optimal binding that is possible within *stk*.

E. Constructing extended framework materials

Extended framework materials are two- or three-dimensional structures that can be periodic and, hence, represented by an infinitely repeating unit cell. Covalent organic frameworks (COFs) and metal–organic frameworks (MOFs) are two classes of extended framework materials that generally use the reticular chemistry approach[60,61] of constructing structures based on a given topology and systematically replacing building blocks on this topology to produce a vast potential chemical space. This process has been replicated in *stk*, and we have introduced the handling of periodicity such that *stk* can construct MOFs and COFs.[62,63] Currently, we focus on 2D-COF construction, but because any **BuildingBlock** instance (with the appropriate number of functional groups) can be placed on any topology, it is possible to construct MOFs by placing metal-complexes on a COF topology.

Within *stk*, we distinguish between discrete molecules and extended materials at the topology level, where we provide a series of topology graphs defined by repeating unit cells for generating structures of extended materials. To construct an extended material, the topology graph's unit cell can be repeated in the *x*, *y*, and *z* directions to create a larger graph that the building blocks are placed on. This approach allows for the creation of infinitely repeating structures, where "periodic" bonds are created at the cell boundaries, or finite structures with unreacted groups at the cell boundaries. This distinction is required to provide an interface for generating crystal structures and "island" models of extended materials [Fig. 12(b)].

At the moment, *stk* contains four common two-dimensional topologies of COFs [hexagonal (net: **hxl**), honeycomb (two variations exist with and without a ditopic linker between three-coordinate nodes; net: **hcb**), square (net: **sql**), and kagome (net: **kgm**)] [Fig. 12(c)]. However, the current implementation is extendable to three-dimensions and other extended topology graphs. For two-dimensional systems, the construction places the COF layer in the *xy* plane and orients building blocks by rotating them along the *z* direction. As with other topology graphs, unsymmetrical building blocks can be manually oriented with respect to their neighbors as desired.

F. Hierarchical construction and further analysis of *stk* molecules

An important feature of *stk* is the easy conversion of a **ConstructedMolecule** into a **BuildingBlock**, which allows for the simple use of a previously constructed molecule as the building block in a new construction as part of a "hierarchical" construction process. Specifically, a **BuildingBlock** can be created from an existing **BuildingBlock** object, or from a **ConstructedMolecule**, but with a different set of functional groups. Therefore, *stk* is well suited for constructing complex molecules over many steps from elementary building blocks. In particular, the **Linear** class, for example, can be used to construct combinatorial libraries of precursor molecules for further construction, e.g., to act as a family of potential rotaxane

(a)

```
1  # Construct 2 * 2 * 1 island structure.
2  topology_graph = stk.cof.Honeycomb(
3      building_blocks=(bb1, bb2),
4      lattice_size=(2, 2, 1),
5  )
6  cof = stk.ConstructedMolecule(topology_graph)
7  stk.MolWriter().write(
8      molecule=cof,
9      path='output/cof_island.mol',
10 )
11
12 # Construct periodic structures.
13 topology_graph = stk.cof.PeriodicHoneycomb(
14     building_blocks=(bb1, bb2),
15     lattice_size=(1, 1, 1),
16 )
17 construction_result = topology_graph.construct()
18 cof = stk.ConstructedMolecule.init_from_construction_result(
19     construction_result=construction_result,
20 )
21 stk.PdbWriter().write(
22     molecule=cof,
23     path='output/cof_periodic.pdb',
24     periodic_info=construction_result.get_periodic_info(),
25 )
```

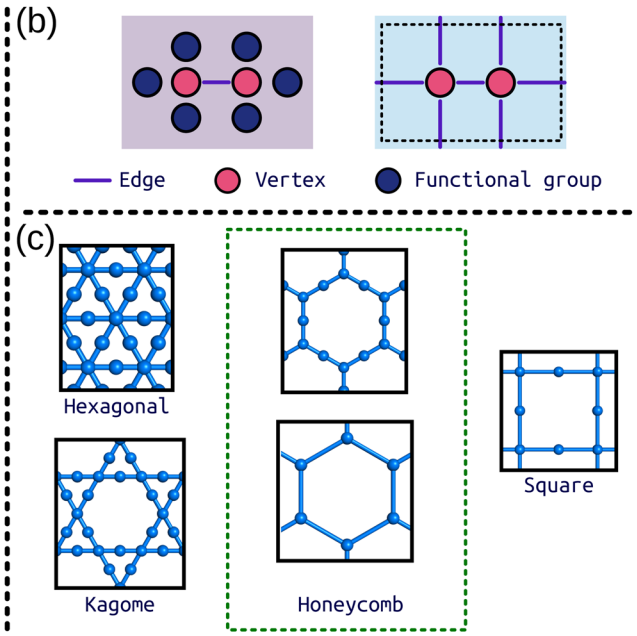


FIG. 12. (a) Assembly of a cluster ("island") and periodic **COF** model. The code snippet highlights the use of a "periodic" **TopologyGraph** to include periodic information, which can now be saved to Protein Data Bank (PDB) or Turbomole files (lines 21–25 of the code snippet). (b) Schematic showing the topological difference between the (left) nonperiodic and (right) periodic cases, where the main difference is the connectivity of the **FunctionalGroup**s at the cell boundaries. (c) Implemented extended topology graphs showing a unit cell containing vertices and edges. Colored boxes in (a) match up to code snippets with their structure in (b). Code snippets from online examples are not complete and require initialization of precursors.

© Author(s) 2021

axles (Fig. 13). Additionally, we find this approach useful for assembling metallo-architectures from **MetalComplex** structures. For example, constructing an **Octahedral** metal complex, and then placing that, as a building block, on the node of a cage topology graph is much simpler than constructing both simultaneously. By tackling the problem in a step-wise fashion, this approach simplifies the underlying topology graph (improving the code's generalizability) and the complexity of the user input (i.e., the building blocks are simpler). A similar approach could be used to construct coordination polymers using the **MetalComplex** and **Linear** classes. Finally, all molecules generated by *stk* can be used to generate new molecules of arbitrary complexity.

In the latest version of *stk*, we also support the writing of molecules to various common file types [XYZ, MOL V3000, Protein Data Bank (PDB), and Turbomole files]. The new implementation can also handle the output of periodic structures [Fig. 12(a)] for relevant file types (namely, PDB and Turbomole files). Users may also straightforwardly define new functions for writing molecules to files if the built-in formats do not match their requirements. Furthermore, *stk* **Molecule** instances can be converted directly into the molecular representation of the cheminformatics software *RDKit*.[37] Ultimately, this allows for the interfacing of molecules generated by *stk* with many other computational chemistry software.

```
macrocycles = [...]
caps = [...]
linears_1 = [...]
linears_2 = [...]

for cap, l1, l2 in product(caps, linears_1, linears_2):
    # Build axle.
    polymer = stk.ConstructedMolecule(
        topology_graph=stk.polymer.Linear(
            building_blocks=(cap, l1, l2),
            repeating_unit='ABCBA',
            num_repeating_units=1,
            orientations=(0, 0, 0, 0, 0),
        ),
    )
    # Convert to building block.
    axle = stk.BuildingBlock.init_from_molecule(polymer)

    # Build all possible rotaxanes.
    for cycle in macrocycles:
        rotaxane = stk.ConstructedMolecule(
            topology_graph=stk.rotaxane.NRotaxane(
                axle=axle,
                cycles=(cycle, ),
                repeating_unit='A',
                num_repeating_units=1,
                orientations=(0.0, ),
            ),
        )
```

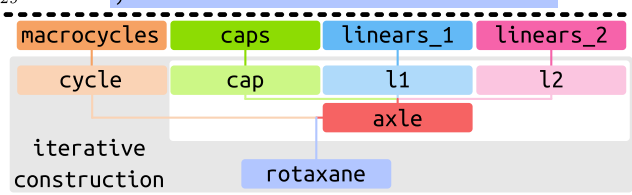


FIG. 13. Code snippet showing the hierarchical assembly of multiple rotaxane structures from a pool of macrocycles, caps, and linear building blocks. Here, "for" loops are used to iterate over these pools of building blocks. Colored boxes match up to code snippets with the hierarchical process schematic below the dashed line.

IV. CONCLUSIONS

stk is a Python library designed for the automated construction of structures of arbitrary complexity from their constituent building blocks. We provide a modular and open-source framework that is generally applicable and simple to extend to a user's material of interest. Currently, constructable molecule types include linear polymers, small molecule oligomers, macrocycles, rotaxanes, metal complexes, metal–organic cages, organic cages, and extended framework materials. Importantly, we provide a robust interface to the methods required for construction: (1) functional group searching; (2) placement and alignment of building blocks on the vertices of a topology graph; and (3) reacting functional groups along edges of the topology graph, all of which can be used in any new user problem and as parts of much larger workflows. When coupled with other open-source codes in the *stk* ecosystem, including *stk-vis* and *stko*, we provide a solution to structure generation and exploration that includes a modular evolutionary algorithm and a simple interface to databasing tools (such as MongoDB) for the simplified storing and sharing of large chemical libraries. With *stk-vis*, the *stk* ecosystem is ideal for real-time collaboration between experimental and computational chemists in various materials chemistry fields. As the active developers of *stk*, we aim to provide consistent improvements and guidance for new users with example usages and tutorials; additionally, we are active in assisting new users in implementing new topology graphs or reactions. Furthermore, *stk* comes with a test-suite covering the code base, which makes extending *stk* simpler and safer. We anticipate that *stk* will provide users with a robust and general solution to structure generation and the pre- and post-processing of structures and precursor generation.

AUTHORS' CONTRIBUTIONS

L.T. is the primary author of the *stk* code. A.T. is the primary author of this manuscript and contributed the metal-complex and metal–organic cage code. F.S. contributed the rotaxane and macrocycle code. All code contributions are overseen by L.T., and contributions are documented on the github history. K.E.J. supervised the development of *stk* and the writing of this manuscript. All authors contributed to the manuscript and oversaw its final production.

ACKNOWLEDGMENTS

K.E.J. acknowledges the Royal Society for a University Research Fellowship and a Royal Society Enhancement Award 2018 (AT), and the ERC through Agreement No. 758370 (ERC-StG-PE5-CoMMaD). We also acknowledge the Leverhulme Trust for a Research Project Grant (FTS). Steven Bennett is thanked for contributions to *stk* and the *stko* code. Dr. Alejandro Santana-Bonilla is thanked for assistance with implementing the xTB wrapper into the *stko* code.

© Author(s) 2021

DATA AVAILABILITY

No data were generated for this manuscript. All code is open-source and linked to throughout the manuscript.

REFERENCES

[1] R. L. Greenaway and K. E. Jelfs, "Integrating computational and experimental workflows for accelerated organic materials discovery," Adv. Mater. **33**, 2004831 (2021).

[2] D. Ongari, L. Talirz, and B. Smit, "Too many materials and too many applications: An experimental problem waiting for a computational solution," ACS Cent. Sci. **6**, 1890–1900 (2020).

[3] R. Pollice, G. dos Passos Gomes, M. Aldeghi, R. J. Hickman, M. Krenn, C. Lavigne, M. Lindner-D'Addario, A. Nigam, C. T. Ser, Z. Yao, and A. Aspuru-Guzik, "Data-driven strategies for accelerated materials design," Acc. Chem. Res. **54**, 849–860 (2021).

[4] K. T. Butler, D. W. Davies, H. Cartwright, O. Isayev, and A. Walsh, "Machine learning for molecular and materials science," Nature **559**, 547–555 (2018).

[5] A. Nandy, C. Duan, J. P. Janet, S. Gugler, and H. J. Kulik, "Strategies and software for machine learning accelerated discovery in transition metal chemistry," Ind. Eng. Chem. Res. **57**, 13973–13986 (2018).

[6] J. P. Janet, S. Ramesh, C. Duan, and H. J. Kulik, "Accurate multiobjective design in a space of millions of transition metal complexes with neural-network-driven efficient global optimization," ACS Cent. Sci. **6**, 513–524 (2020).

[7] J. P. Janet, C. Duan, A. Nandy, F. Liu, and H. J. Kulik, "Navigating transition-metal chemical space: Artificial intelligence for first-principles design," Acc. Chem. Res. **54**, 532–545 (2021).

[8] R. Gómez-Bombarelli, J. Aguilera-Iparraguirre, T. D. Hirzel, D. Duvenaud, D. Maclaurin, M. A. Blood-Forsythe, H. S. Chae, M. Einzinger, D.-G. Ha, T. Wu, G. Markopoulos, S. Jeon, H. Kang, H. Miyazaki, M. Numata, S. Kim, W. Huang, S. I. Hong, M. Baldo, R. P. Adams, and A. Aspuru-Guzik, "Design of efficient molecular organic light-emitting diodes by a high-throughput virtual screening and experimental approach," Nat. Mater. **15**, 1120–1127 (2016).

[9] L. Wilbraham, R. S. Sprick, K. E. Jelfs, and M. A. Zwijnenburg, "Mapping binary copolymer property space with neural networks," Chem. Sci. **10**, 4973–4984 (2019).

[10] M. Miklitz, S. Jiang, R. Clowes, M. E. Briggs, A. I. Cooper, and K. E. Jelfs, "Computational screening of porous organic molecules for xenon/krypton separation," J. Phys. Chem. C **121**, 15211–15222 (2017).

[11] M. Tong, Y. Lan, Z. Qin, and C. Zhong, "Computation-ready, experimental covalent organic framework for methane delivery: Screening and material design," J. Phys. Chem. C **122**, 13009–13016 (2018).

[12] D. Ongari, A. V. Yakutovich, L. Talirz, and B. Smit, "Building a consistent and reproducible database for adsorption evaluation in covalent–organic frameworks," ACS Cent. Sci. **5**, 1663–1675 (2019).

[13] Z. Yao, B. Sánchez-Lengeling, N. S. Bobbitt, B. J. Bucior, S. G. H. Kumar, S. P. Collins, T. Burns, T. K. Woo, O. K. Farha, R. Q. Snurr, and A. Aspuru-Guzik, "Inverse design of nanoporous crystalline reticular materials with deep generative models," Nat. Mach. Intell. **3**, 76–86 (2021).

[14] B. P. Hay and T. K. Firman, "HostDesigner: A program for the de novo structure-based design of molecular receptors with binding sites that complement metal ion guests," Inorg. Chem. **41**, 5502–5512 (2002).

[15] T. A. Young, R. Gheorghe, and F. Duarte, "cgbind: A python module and web app for automated metallocage construction and host–guest characterization," J. Chem. Inf. Model. **60**, 3546–3557 (2020).

[16] E. I. Ioannidis, T. Z. H. Gani, and H. J. Kulik, "molSimplify: A toolkit for automating discovery in inorganic chemistry," J. Comput. Chem. **37**, 2106–2117 (2016).

[17] J.-G. Sobez and M. Reiher, "Molassembler: Molecular graph construction, modification, and conformer generation for inorganic and organic molecules," J. Chem. Inf. Model. **60**, 3884–3900 (2020).

[18] L. J. Abbott, K. E. Hart, and C. M. Colina, "Polymatic: A generalized simulated polymerization algorithm for amorphous polymers," Theor. Chem. Acc. **132**, 1334 (2013).

[19] C. E. Wilmer, M. Leaf, C. Y. Lee, O. K. Farha, B. G. Hauser, J. T. Hupp, and R. Q. Snurr, "Large-scale screening of hypothetical metal–organic frameworks," Nat. Chem. **4**, 83–89 (2012).

[20] P. G. Boyd and T. K. Woo, "A generalized method for constructing hypothetical nanoporous materials of any net topology from graph theory," CrystEngComm **18**, 3777–3792 (2016).

[21] Y. J. Colón, D. A. Gómez-Gualdrón, and R. Q. Snurr, "Topologically guided, automated construction of metal–organic frameworks and their evaluation for energy-related applications," Cryst. Growth Des. **17**, 5801–5810 (2017).

[22] R. L. Martin and M. Haranczyk, "Construction and characterization of structure models of crystalline porous polymers," Cryst. Growth Des. **14**, 2431–2440 (2014).

[23] J. Keupp and R. Schmid, "TopoFF: MOF structure prediction using specifically optimized blueprints," Faraday Discuss. **211**, 79–101 (2018).

[24] M. A. Addicoat, D. E. Coupry, and T. Heine, "AuToGraFS: Automatic topological generator for framework structures," J. Phys. Chem. A **118**, 9607–9614 (2014).

[25] J. P. Darby, M. Arhangelskis, A. D. Katsenis, J. M. Marrett, T. Friščić, and A. J. Morris, "*Ab initio* prediction of metal–organic framework structures," Chem. Mater. **32**, 5835–5844 (2020).

[26] P. G. Boyd, Y. Lee, and B. Smit, "Computational development of the nanoporous materials genome," Nat. Rev. Mater. **2**, 17037 (2017).

[27] L. Turcani, E. Berardo, and K. E. Jelfs, "*stk*: A python toolkit for supramolecular assembly," J. Comput. Chem. **39**, 1931–1942 (2018).

[28] L. Turcani, stk-vis, https://github.com/lukasturcani/stk-vis; accessed February 18, 2021.

[29] S. Bennett, A. Tarzia, and L. Turcani, stko: stk-optimizers, https://github.com/JelfsMaterialsGroup/stko; accessed February 18, 2021.

[30] A. H. Larsen, J. J. Mortensen, J. Blomqvist, I. E. Castelli, R. Christensen, M. Dułak, J. Friis, M. N. Groves, B. Hammer, C. Hargus, E. D. Hermes, P. C. Jennings, P. B. Jensen, J. Kermode, J. R. Kitchin, E. L. Kolsbjerg, J. Kubal, K. Kaasbjerg, S. Lysgaard, J. B. Maronsson, T. Maxson, T. Olsen, L. Pastewka, A. Peterson, C. Rostgaard, J. Schiøtz, O. Schütt, M. Strange, K. S. Thygesen, T. Vegge, L. Vilhelmsen, M. Walter, Z. Zeng, and K. W. Jacobsen, "The atomic simulation environment—A python library for working with atoms," J. Phys.: Condens. Matter **29**, 273002 (2017).

[31] P. Eastman, J. Swails, J. D. Chodera, R. T. McGibbon, Y. Zhao, K. A. Beauchamp, L.-P. Wang, A. C. Simmonett, M. P. Harrigan, C. D. Stern, R. P. Wiewiora, B. R. Brooks, and V. S. Pande, "OpenMM 7: Rapid development of high performance algorithms for molecular dynamics," PLOS Comput. Biol. **13**, e1005659 (2017).

[32] S. P. Ong, W. D. Richards, A. Jain, G. Hautier, M. Kocher, S. Cholia, D. Gunter, V. L. Chevrier, K. A. Persson, and G. Ceder, "Python materials genomics (pymatgen): A robust, open-source python library for materials analysis," Comput. Mater. Sci. **68**, 314–319 (2013).

[33] Schrödinger release 2021-1: MacroModel (Schrödinger, LLC, New York, 2021).

[34] J. D. Gale, "GULP: A computer program for the symmetry-adapted simulation of solids," J. Chem. Soc., Faraday Trans. **93**, 629–637 (1997).

[35] J. D. Gale and A. L. Rohl, "The general utility lattice program (GULP)," Mol. Simul. **29**, 291–341 (2003).

[36] C. Bannwarth, S. Ehlert, and S. Grimme, "GFN2-xTB—An accurate and broadly parametrized self-consistent tight-binding quantum chemical method with multipole electrostatics and density-dependent dispersion contributions," J. Chem. Theory Comput. **15**, 1652–1671 (2019).

[37] G. A. Landrum, RDKit: Open-source cheminformatics, http://rdkit.org/; accessed March 1, 2020.

[38] N. Brown, B. McKay, F. Gilardoni, and J. Gasteiger, "A graph-based genetic algorithm and its application to the multiobjective evolution of median molecules," J. Chem. Inf. Comput. Sci. **44**, 1079–1087 (2004).

[39] N. M. O'Boyle, C. M. Campbell, and G. R. Hutchison, "Computational design and selection of optimal organic photovoltaic materials," J. Phys. Chem. C **115**, 16200–16210 (2011).

[40] I. Y. Kanal and G. R. Hutchison, "Rapid computational optimization of molecular properties using genetic algorithms: Searching across millions of

© Author(s) 2021

compounds for organic photovoltaic materials," arXiv:1707.02949 [physics] (2017).
[41] J. H. Jensen, "A graph-based genetic algorithm and generative model/Monte Carlo tree search for the exploration of chemical space," Chem. Sci. **10**, 3567–3572 (2019).
[42] E. Berardo, L. Turcani, M. Miklitz, and K. E. Jelfs, "An evolutionary algorithm for the discovery of porous organic cages," Chem. Sci. **9**, 8513–8527 (2018).
[43] M. Miklitz, L. Turcani, R. L. Greenaway, and K. E. Jelfs, "Computational discovery of molecular C_{60} encapsulants with an evolutionary algorithm," Commun. Chem. **3**, 10 (2020).
[44] L. Wilbraham, E. Berardo, L. Turcani, K. E. Jelfs, and M. A. Zwijnenburg, "High-throughput screening approach for the optoelectronic properties of conjugated polymers," J. Chem. Inf. Model. **58**, 2450–2459 (2018).
[45] R. S. Sprick, C. M. Aitchison, E. Berardo, L. Turcani, L. Wilbraham, B. M. Alston, K. E. Jelfs, M. A. Zwijnenburg, and A. I. Cooper, "Maximising the hydrogen evolution activity in organic photocatalysts by co-polymerisation," J. Mater. Chem. A **6**, 11994–12003 (2018).
[46] I. Heath-Apostolopoulos, L. Wilbraham, and M. A. Zwijnenburg, "Computational high-throughput screening of polymeric photocatalysts: Exploring the effect of composition, sequence isomerism and conformational degrees of freedom," Faraday Discuss. **215**, 98–110 (2019).
[47] Y. Bai, L. Wilbraham, B. J. Slater, M. A. Zwijnenburg, R. S. Sprick, and A. I. Cooper, "Accelerated discovery of organic polymer photocatalysts for hydrogen evolution from water through the integration of experiment and theory," J. Am. Chem. Soc. **141**, 9063–9071 (2019).
[48] C. B. Meier, R. Clowes, E. Berardo, K. E. Jelfs, M. A. Zwijnenburg, R. S. Sprick, and A. I. Cooper, "Structurally diverse covalent triazine-based framework materials for photocatalytic hydrogen evolution from water," Chem. Mater. **31**, 8830–8838 (2019).
[49] I. Heath-Apostolopoulos, D. Vargas-Ortiz, L. Wilbraham, K. E. Jelfs, and M. Zwijnenburg, "Using high-throughput virtual screening to explore the optoelectronic property space of organic dyes; finding diketopyrrolopyrrole dyes for dye-sensitized water splitting and solar cells," Sustainable Energy Fuels **5**, 704–719 (2020).
[50] T. Hasell and A. I. Cooper, "Porous organic cages: Soluble, modular and molecular pores," Nat. Rev. Mater. **1**, 16053 (2016).
[51] T. R. Cook and P. J. Stang, "Recent developments in the preparation and chemistry of metallacycles and metallacages via coordination," Chem. Rev. **115**, 7001–7045 (2015).
[52] B. S. Pilgrim and N. R. Champness, "Metal–organic frameworks and metal–organic cages—A perspective," ChemPlusChem **85**, 1842–1856 (2020).
[53] H. Vardhan, M. Yusubov, and F. Verpoort, "Self-assembled metal–organic polyhedra: An overview of various applications," Coord. Chem. Rev. **306**, 171–194 (2016).
[54] E. Berardo, R. L. Greenaway, L. Turcani, B. M. Alston, M. J. Bennison, M. Miklitz, R. Clowes, M. E. Briggs, A. I. Cooper, and K. E. Jelfs, "Computationally-inspired discovery of an unsymmetrical porous organic cage," Nanoscale **10**, 22381–22388 (2018).
[55] L. Turcani, R. L. Greenaway, and K. E. Jelfs, "Machine learning for organic cage property prediction," Chem. Mater. **31**, 714–727 (2019).
[56] V. Santolini, M. Miklitz, E. Berardo, and K. E. Jelfs, "Topological landscapes of porous organic cages," Nanoscale **9**, 5280–5298 (2017).
[57] T. A. Young, V. Martí-Centelles, J. Wang, P. J. Lusby, and F. Duarte, "Rationalizing the activity of an 'artificial Diels-Alderase': Establishing efficient and accurate protocols for calculating supramolecular catalysis," J. Am. Chem. Soc. **142**, 1300–1310 (2020).
[58] M. Xue, Y. Yang, X. Chi, X. Yan, and F. Huang, "Development of pseudorotaxanes and rotaxanes: From synthesis to stimuli-responsive motions to applications," Chem. Rev. **115**, 7398–7501 (2015).
[59] S. Zarra, D. M. Wood, D. A. Roberts, and J. R. Nitschke, "Molecular containers in complex chemical systems," Chem. Soc. Rev. **44**, 419–432 (2015).
[60] O. M. Yaghi, M. O'Keeffe, N. W. Ockwig, H. K. Chae, M. Eddaoudi, and J. Kim, "Reticular synthesis and the design of new materials," Nature **423**, 705–714 (2003).
[61] H. Furukawa, K. E. Cordova, M. O'Keeffe, and O. M. Yaghi, "The chemistry and applications of metal–organic frameworks," Science **341**, 1230444 (2013).
[62] J. D. Crowley, S. M. Goldup, A.-L. Lee, D. A. Leigh, and R. T. McBurney, "Active metal template synthesis of rotaxanes, catenanes and molecular shuttles," Chem. Soc. Rev. **38**, 1530–1541 (2009).
[63] P. Z. Moghadam, T. Islamoglu, S. Goswami, J. Exley, M. Fantham, C. F. Kaminski, R. Q. Snurr, O. K. Farha, and D. Fairen-Jimenez, "Computer-aided discovery of a metal–organic framework with superior oxygen uptake," Nat. Commun. **9**, 1378 (2018).

© Author(s) 2021