MOF-Adsorbate-Initializer Documentation

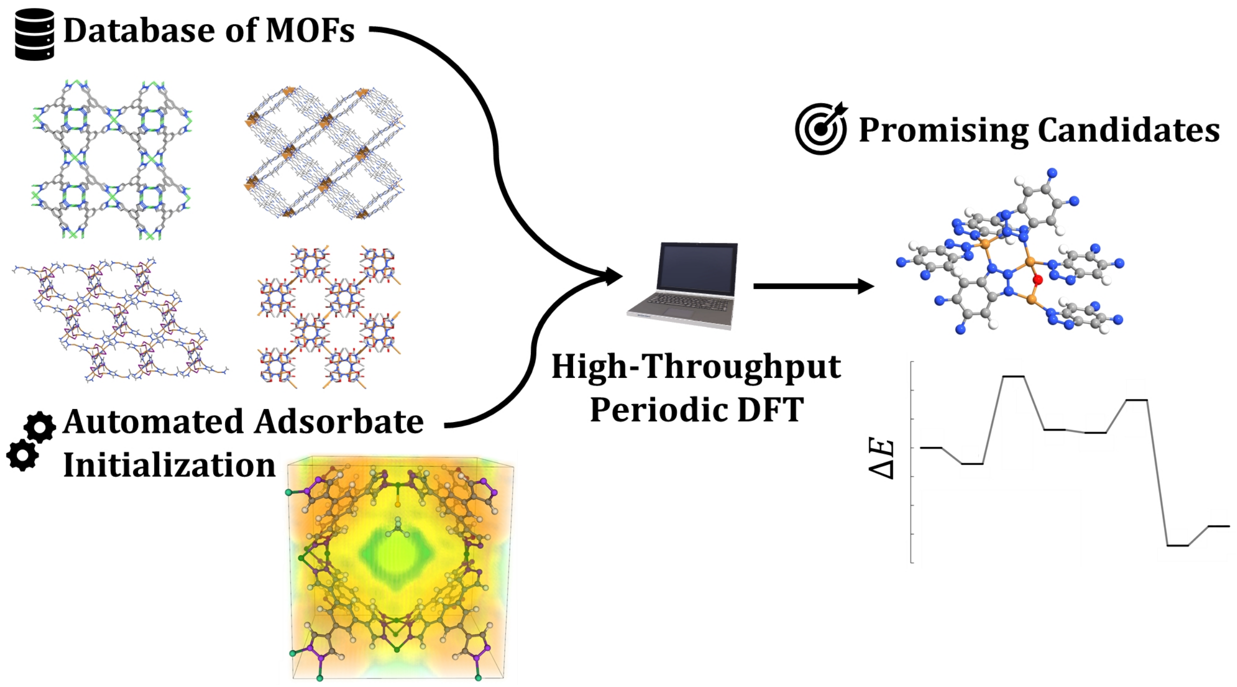
Release v1.0

Andrew S. Rosen

Jan 10, 2019

Contents

The MOF Adsorbate Initializer (MAI) is a set of Python tools to initialize the positions of small molecule adsorbates at coordinatively unsaturated sites in metal-organic frameworks (MOFs) in a high-throughput, fully automated manner. The main benefit of MAI is the ability to systematically construct initial structures for large-scale density functional theory (DFT) calculations on many hundreds of materials. With a dataset of initial structures, workflow tools like PyMOFScreen or Atomate can be used to run the structural optimizations with your favorite optimzier of choice.


Database of MOFs
Promising Candidates
High-Throughput Periodic DFT
Automated Adsorbate Initialization
ΔE

CHAPTER 1

Contents

1.1 Installation

The MAI code can be found on GitHub and can be downloaded from the webpage or cloned via `git clone git@github.com:arosen93/mof-adsorbate-initializer.git`.

1.1.1 Dependencies

1. Python 3.6 or newer
2. Pymatgen 2018.11.30 or newer
3. ASE 3.16.0 or newer
4. OpenMetalDetector (recommended)

1.1.2 Installation Instructions

1. If you don't already have Python installed on your machine, you'll need to install Python 3.6 or newer. I recommend using the Anaconda distribution of Python 3 if you don't already have it installed.
2. You will then need to install MAI and all of the required dependencies. The easiest way to do this is to find the `requirements.txt` file in the MAI base directory and use the command `pip install -r requirements.txt; pip install .`.

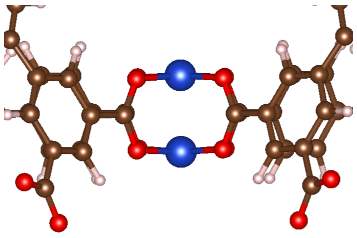
1.2 Tutorials

It is probably best to lead by example, so in this tutorial I will walk you through a few examples meant to highlight MAI's capabilities and get you up to speed. Each one progressively builds on the one prior, so if you're new around here, I recommend you start at the top. Let's get started! In addition to the tutorials shown here, please refer to the `examples` folder in the main MAI directory for example scripts you can run and play around with.

1.2.1 Contents

Monatomics

In this example, we'll work through how to add a single atom adsorbate to open metal site in MOFs. The CIF for the MOF we'll use in this example can be found `here`. This MOF is known as Cu-BTC and has the structure shown below:



The metal (Cu) sites here are shown in orange. There are multiple Cu sites per unit cell, and each Cu site is in a paddlewheel-like structure. For this example, we will consider the initialization of an O atom adsorbate to a single coordinatively unsaturated Cu site.

We'll start with the code that can do the job. Then we'll walk through what it all means.

```
import os
from mai.adsorbate_constructor import adsorbate_constructor

mof_path = os.path.join('example_MOFs','Cu-BTC.cif') #path to CIF of MOF

#add O adsorbate to index 0
ads = adsorbate_constructor(ads='O',d_MX1=1.75)
new_mof_atoms = ads.get_adsorbate(atoms_path=mof_path,site_idx=0)
```

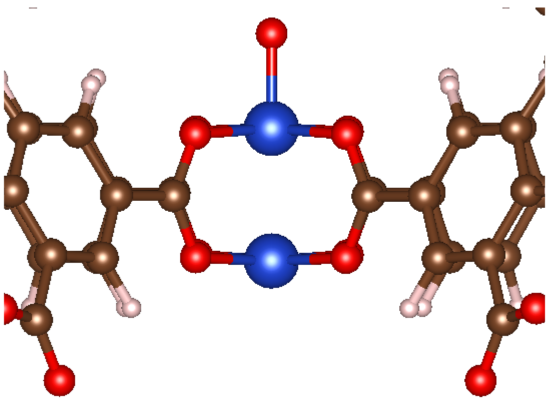
Okay, let's dive right in! MAI requires the calling of an object known as the *`adsorbate_constructor`*, which tells MAI what kind of adsorbate you'd like to make. For simple monatomic species, there are only a few arguments you need to worry about.

1. The `ads` keyword argument is a string of the element that you want to add to the structure.
2. The `d_MX1` keyword argument is the desired distance between the adsorption site (i.e. the Cu species) and the adsorbate (in Å).

That takes care of initializing the *`adsorbate_constructor`* object. Now we can use this object to call a function to initialize the adsorbate. This is done via *`get_adsorbate()`*. The output of calling *`get_adsorbate()`* is a new ASE `Atoms` object with the adsorbate initialized. The commonly used keywords for monatomic speies are as follows:

1. The `atoms_path` keyword argument is the filepath to the starting CIF file of the MOF.
2. The `site_idx` keyword argument is an integer representing the ASE `Atoms` index of the adsorption site (i.e. the Cu species). Later in this guide we'll show how this parameter can be determined automatically, but for now we have manually set it to the 0-th `Atoms` index via `site_idx=0`, which corresponds to one of the Cu atoms. To find out the ASE indices for a given structure, you can inspect or visualize the `Atoms` object associated with the CIF file. Generally, it is the same indexing order as you'd find in your favorite CIF viewer (e.g. VESTA).

Now let's see what happens as a result of running this code! The initialized structure is shown below:



Exactly what we'd expect! Generally, MAI aims to satisfy two major conditions. The first condition is that it tries to maximize the symmetry of the first coordination sphere when the adsorbate is added. In this case, the geometry is square planar prior to adsorption, so MAI makes a square pyramidal structure when the monatomic species is added. The second condition is that MAI tries to minimize steric interactions when possible. In the case of a paddlewheel structure like Cu-BTC, the monatomic adsorbate could have been initialize in one of two directions normal to the square planar first coordination sphere. However, only one of those directions is geometrically accessible (the other is pointed inward between the Cu paddlewheel, which would not be reasonable).

That concludes our tutorial with monatomic adsorbates. Join me as we move onto more complicated systems! Up next is diatomics!

Diatomics

In this example, we'll work through how to add two-atom adsorbates to open metal sites in MOFs. The CIF for the MOF we'll use for this example can be found `here`. This MOF is known Ni3(BTP)2, or Ni-BTP for short, and has a sodalite-like structure with square planar Ni cations.

Homoatomic

For this example, we will consider the initialization of an O2 molecule to a single coordinatively unsaturated Ni site. O2 can bind in an end-on (η1-O) or side-on (η2-O) mode depending on the structure. We'll consider both for this example. The code to handle this is shown below.

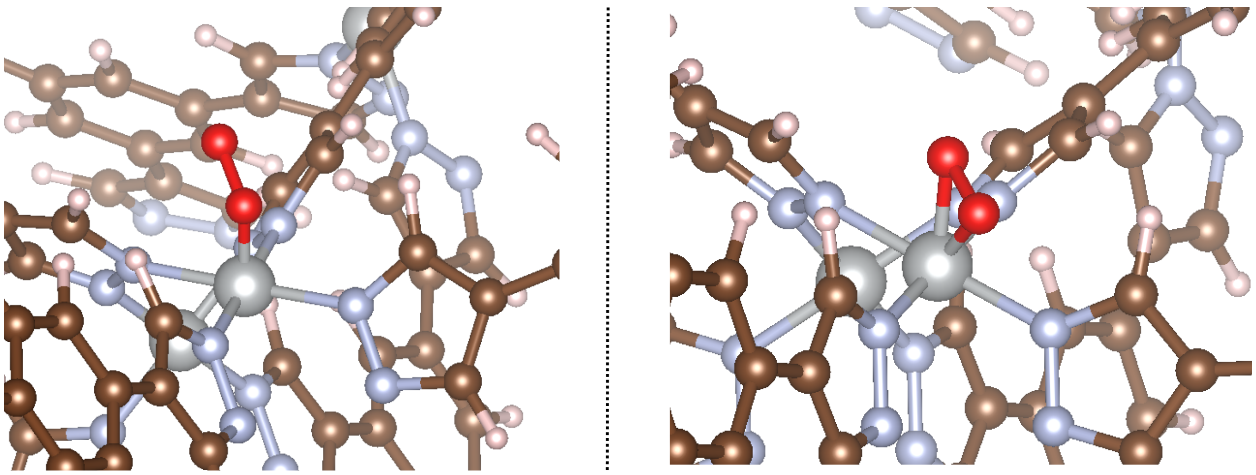
```python
import os
from mai.adsorbate_constructor import adsorbate_constructor

mof_path = os.path.join('example_MOFs','Ni-BTP.cif') #path to CIF of MOF

#add O2 adsorbate in η1-O mode
ads = adsorbate_constructor(ads='O2_end',d_MX1=1.5,eta=1,
        d_X1X2=1.2,ang_MX1X2=120)
new_mof_atoms1 = ads.get_adsorbate(atoms_path=mof_path,site_idx=0)

#add O2 adsorbate in η2-O mode
ads = adsorbate_constructor(ads='O2_side',d_MX1=1.5,eta=2,
        d_X1X2=1.2,ang_MX1X2=90)
new_mof_atoms2 = ads.get_adsorbate(atoms_path=mof_path,site_idx=0)
```

Like with the monatomic example, we need to initialize an *adsorbate_constructor* object and then provide it the MOF of interest. In the case of diatomics, we have a few new keywords to introduce. In addition to the arguments described in the monatomic tutorial, we now need to be able to tell MAI what kind of denticity we would like (i.e.

end-on or side-on adsorption) and what we want the X1-X2 bond length and M-X1-X2 bond angle to be (if X1-X2 is our diatomic of interest and M is our metal adsorption site). The arguments used here are described below:

1. The `ads` argument is a string of the molecule that you want to add to the structure. Note that MAI will internally strip any characters following (and including) an underscore, so `ads='O2_end'` and `ads='O2_side'` both get stripped to 'O2'. That being said, the full string for the `ads` argument will be used when writing the filenames of the new CIFs, so using an underscore can be helpful for organizational purposes.
2. The `d_MX1` argument is the desired distance between the adsorption site (i.e. the Ni species) and the adsorbate (in Å). If the adsorbate is bound in an end-on fashion, this represents the M-X1 distance. If the adsorbate is bound in a side-on fashion, this represents the distance between M and the midpoint between X1 and X2. Here, we set `d_MX1=1.5`.
3. The `eta` keyword argument is an integer representing the denticity. In other words, `eta=1` would be an end-on adsorption mode, whereas `eta=2` would be a side-on adsorption mode. By default, `eta=1` if unspecified. For this example, we decided to explore both options.
4. The `d_X1X2` keyword argument is the desired distance between X1 and X2 (in Å). If not specified, it will default to the value for `d_MX1`. Here, we decided to set `d_X1X2=1.2`, which is a reasonable O-O bond distance.
5. The `ang_MX1X2` keyword argument is the angle between the adsorption site and the adsorbate (in degrees). If the adsorbate is bound in an end-on fashion, this represents the M-X1-X2 bond angle. If the adsorbate is bound in a side-on fashion, this represents the angle between M, the midpoint between X1 and X2, and X2. By default, it assumes `ang_MX1X2=180` if `eta=1` and `ang_M1X2=90` if `eta=2`. For this example, we use `ang_MX1X2=120` and `ang_MX1X2=90`, respectively, which is representative of common O2 binding geometries.

That takes care of initializing the *`adsorbate_constructor`* object. With this, we provide the object with the path to the MOF and the site index, and it will initialize the adsorbate for us. Now let's see what happens as a result of running this code! The initialized structures are shown below:



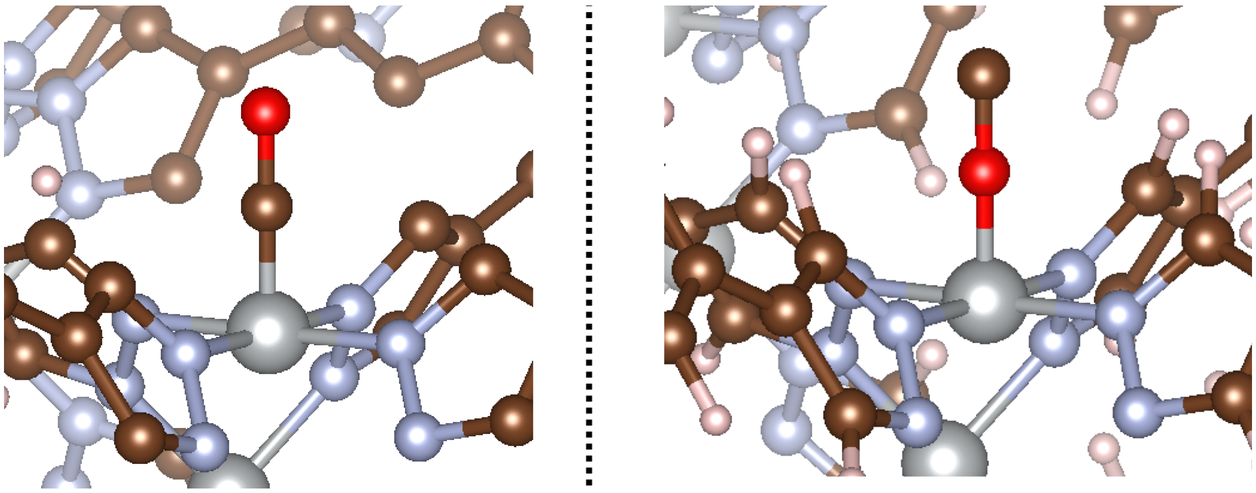
Exactly what we'd expect yet again! You can see that in the first example, O2 is bound end-on, whereas in the second it is bound side-on, as specified in the example script. The bond angles and distances are the same as those specified in the input file.

Heteroatomic

MAI also supports heteratomic adsorbates. In this example, we'll consider the adsorption of a single CO molecule with the same MOF. The only thing that changes for heteroatomic adsorbates is that you need to tell MAI which atom is the "connecting atom" (i.e. the atom of the adsorbate bound to the metal adsorption site) if bound in an end-on fashion. By default, MAI will assume that the first atom in `ads` is the connecting atom. Therefore, setting `ads='CO'` or `ads='OC'` would yield M-C-O or M-O-C binding modes, respectively.

```
import os
from mai.adsorbate_constructor import adsorbate_constructor

mof_path = os.path.join('example_MOFs','Ni-BTP.cif') #path to CIF of MOF

#add CO adsorbate in η1-C mode
ads = adsorbate_constructor(ads='CO',d_MX1=1.5,d_X1X2=1.13)
new_mof_atoms1 = ads.get_adsorbate(atoms_path=mof_path,site_idx=0)

#add CO adsorbate in η1-O mode
ads = adsorbate_constructor(ads='OC',d_MX1=1.5,d_X1X2=1.13)
new_mof_atoms2 = ads.get_adsorbate(atoms_path=mof_path,site_idx=0)
```

The result of running this code is shown below:



That concludes our tutorial for diatomic adsorbates. Now onto triatomics!

Triatomics

In this example, we'll work through how to add three-atom adsorbates to open metal sites in MOFs. The CIF for the MOF we'll use for this example can be found `here`. This MOF is known as MIL-8BB and has trimetallic nodes with each metal cation in a square pyramidal geometry.

Contiguous Adsorbate

For this example, we will consider the initialization of a "contiguous" triatomic adsorbate. What I mean by this is that if we treat the adsorbate as an arbitrary molecule X1-X2-X3, then the adsorption process is described by M-X1-X2-X3. In this example, we'll consider the adsorption of N2O to an Fe site, in both an η1-N and η1-O binding mode. The code to handle this is shown below.

```
import os
from mai.adsorbate_constructor import adsorbate_constructor

mof_path = os.path.join('example_MOFs','Fe-MIL-88B.cif') #path to CIF of MOF

#add N2O adsorbate in η1-N mode
ads = adsorbate_constructor(ads='N2O',d_MX1=2.0,d_X1X2=1.13,ang_MX1X2=180,d_X2X3=1.19)
new_mof_atoms1 = ads.get_adsorbate(atoms_path=mof_path,site_idx=0)
```

(continues on next page)

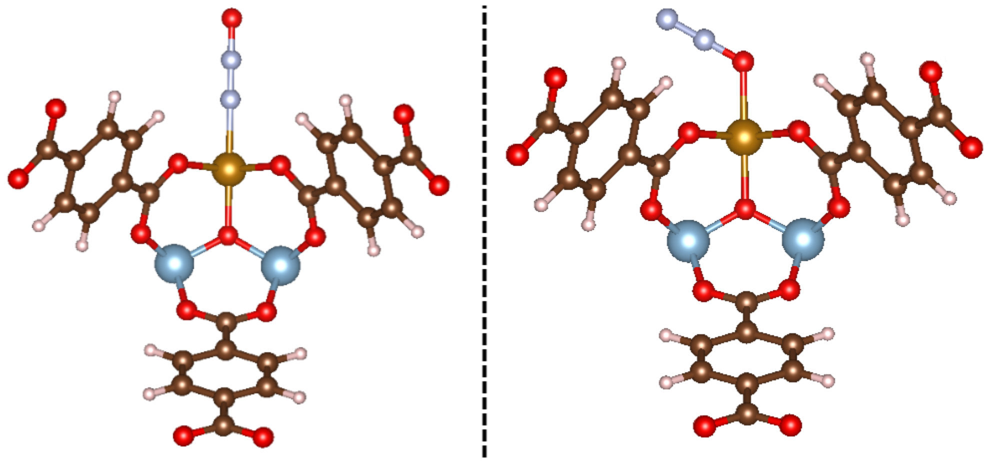
(continued from previous page)

```
#add N2O adsorbate in η1-O mode
ads = adsorbate_constructor(ads='ON2',d_MX1=2.0,d_X1X2=1.19,ang_MX1X2=120,d_X2X3=1.13)
new_mof_atoms2 = ads.get_adsorbate(atoms_path=mof_path,site_idx=0)
```

Like with the previous examples, we need to initialize an `adsorbate_constructor` object and then provide it the MOF of interest. In the case of triatomics, we have a few new keywords to introduce. In addition to the arguments described in the previous examples, we can now provide MAI additional geometric parameters if desired. Namely, the new features are now that we can include the X2-X3 bond length and the X1-X2-X3 bond angle. The arguments used here are described below:

1. The `ads`, `d_MX1X2`, `d_X1X2`, and `ang_MX1X2` are the same as before.
2. Now, we have the option to add the `d_X2X3` keyword argument, which specifies the X2-X3 distance. It defaults to `d_X2X3=d_X1X2` if not specified.
3. We can also add the `ang_triads` keyword argument, which specifies the X1-X2-X3 bond angle. It defaults to `ang_triads=180` if not specified.

That takes care of initializing the `adsorbate_constructor` object. With this, we provide the object with the path to the MOF, and it will initialize the adsorbate. Now let's see what happens as a result of running this code! The initialized structures are shown below:



Exactly what we'd expect once more! You can see that in the first example, N2O is bound in an η1-N mode, whereas the second is bound in an η1-O mode, as specified. Feel free to play around with the bond distance and bond angle arguments to get a feel for how MAI works.

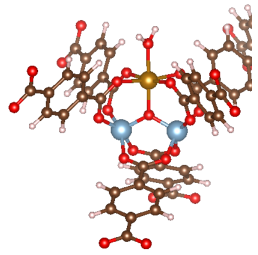
Noncontiguous Adsorbate

The last bit of trickery comes into play when dealing with what I'll call "noncontiguous" adsorbates. These are adsorbates like water, where it is triatomic, but it is not bound in a sequential fashion. As with water, you will have a central atom of the adsorbate bound to the metal (instead of an M-O-H-H adsorption mode). To tell MAI about this desired adsorption mode, `adsorbate_constructor` has a keyword argument named `connect`, which is the atom number in `ads` that should be bound to the adsorption site. For the case of `ads='HOH'`, we should set `connect=2` to have the second atom (i.e. the O atom) bound to the adsorption site. Note that for this reason we cannot use `ads='H2O'` here.

An example code is shown below. The main thing to keep in mind is that now the connecting atom of the adsorbate is X2 instead of X1. If not specified, `ang_triads` will default to `ang_triads=ang_MX1X2` if `connect=2`, but here we set it to 104.5°, as is standard for H2O.

```
import os
from mai.adsorbate_constructor import adsorbate_constructor

mof_path = os.path.join('example_MOFs','Fe-MIL-88B.cif') #path to CIF of MOF

#add H2O adsorbate
ads = adsorbate_constructor(ads='HOH',d_MX1=2.0,d_X1X2=0.96,d_X2X3=0.96,
        ang_MX1X2=120.0,ang_triads=104.5,connect=2)
new_mof_atoms = ads.get_adsorbate(atoms_path=mof_path,site_idx=0)
```

This results in the following initialized structure:



That concludes our tutorial for triatomic adsorbates.

Open Metal Detection

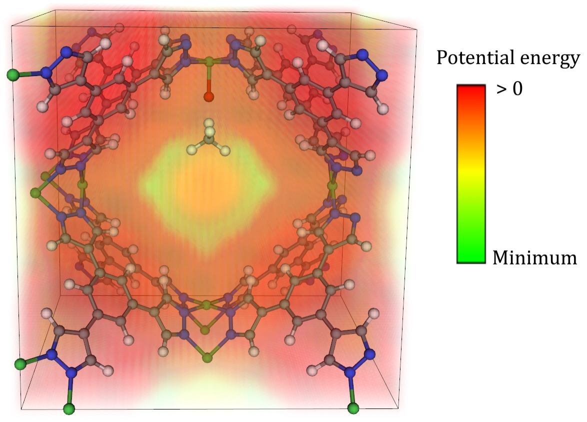
In the previous examples, we manually specified the desired adsorption site via the `site_idx` argument to *get_adsorbate()*. In practice, it is often desirable to have an automated approach for determining which atom in a MOF is the desired adsorption site. By default, if the user does not specify `site_idx`, MAI will assume that you want an automated approach for determining this parameter. By default, MAI will attempt to read in the results from the OpenMetalDetector (OMD) code, which can be used to output information about the number, type, and properties of various open metal sites in a set of MOFs. Instructions for using OMD can be found elsewhere, but the main idea is to call OMD prior to calling MAI. An example is shown below for adding an O atom adsorbate to a bunch of MOFs stored in your current working directory.

```
import os
from omsdetector import MofCollection
from mai.adsorbate_constructor import adsorbate_constructor

mofs_path = 'example_MOFs' #path to folder of CIFs
oms_analysis_folder = os.getcwd() #path to store the OMS results
oms_data_path = os.path.join(oms_analysis_folder,'oms_results') #path to oms_results␣
↪folder

#Run the Open Metal Detector
mof_coll = MofCollection.from_folder(collection_folder=mofs_path,
        analysis_folder=oms_analysis_folder)
mof_coll.analyse_mofs()

#add adsorbate for every CIF in mofs_path
for file in os.listdir(mofs_path):
        if '.cif' not in file:
                continue
        mof_path = os.path.join(mofs_path,file)
        ads = adsorbate_constructor(ads='O',d_MX1=1.75)
        new_mof_atoms = ads.get_adsorbate(atoms_path=mof_path,
                omd_path=oms_data_path)
```

You can see that this workflow is nearly identical to that in the monatomics tutorial, except we did not specify `site_idx` in *`get_adsorbate()`*, and we ran OMD's `MofCollection.from_folder()` command prior to calling MAI's *`adsorbate_constructor`*. Generally, the only arguments you'll need to provide to `MofCollection.from_folder()` is the `collection_folder` and `analysis_folder`, which is where the CIF files are located and where you'd like to store the OMD results, respectively. Then, you'll run OMD's `analyse_mofs()` command to run the analysis. From there, you're ready to run MAI as usual! The only new MAI-related keyword is that *`get_adsorbate()`* can take an `omd_path` keyword argument, which should be where the results from the OMD run are stored. This is generally located at `analysis_folder/oms_results`.

Potential Energy Grids

Some adsorbates do not lend themselves well to the geometric approaches laid out thus far. This is particularly the case for adsorbates that are physisorbed relatively far away from an adsorption site, as opposed to chemisorbed nearby. In these cases, an alternative way of initializing the adsorbate can be considered by mapping out a potential energy grid (PEG) of each MOF and putting the adsorbate in a low-energy site within some cutoff radius of the proposed adsorption site. A visualized example of such a PEG is shown below for a methane adsorbate near a proposed Ni-O active site.



MAI supports two different formats for PEGs. The first is the cube file format, such as that generated from PorousMaterials.jl. Details of how to generate such PEGs can be found here. The second accepted format is a space-delimited file with four columns of (x,y,z,E) entries, where E is the potential energy and (x,y,z) are the coordinates. Each new line represents a new (x,y,z,E) vector. We'll refer to this file as an ASCII grid.

Currently, only single-site CH4 adsorbates are supported with PEGs, although in principle it is trivial to consider other adsorbates as well. When using PEGs to initialize the position of CH4 adsorbates, the C atom of the CH4 molecule will be placed in the low-energy site, and the four remaining H atoms will be arranged to form the tetrahedral structure of CH4, with one of the H atoms pointed directly toward the adsorption site.

Unlike with the monatomics, diatomics, and triatomics tutorials, when dealing with PEGs, one must use the *`get_adsorbate_grid()`* function instead of *`get_adsorbate()`*. The *`get_adsorbate_grid()`* function is quite simple. An example code is shown below to initialize an adsorbate based on an `example PEG` and `example MOF`.

```
import os
from mai.adsorbate_constructor import adsorbate_constructor
from ase.io import read

grid_path = os.path.join('example_MOFs','energy_grids_ASCII') #path to PEG
mof_path = os.path.join(grid_path,'AHOKIR01-O.cif') #path to CIF of MOF

#Select the last O index in the MOF as the adsorption site
```

(continues on next page)

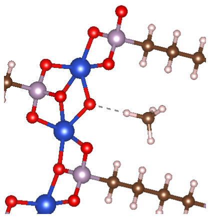
(continued from previous page)

```
mof = read(mof_path)
site_idx = [atom.index for atom in mof if atom.symbol == 'O'][-1]

#Add the CH4 molecule to the O adsorption site within 3 A sphere
ads = adsorbate_constructor(ads='CH4',d_MX1=3.0)
mof_adsorbate = ads.get_adsorbate_grid(atoms_path=mof_path,
        site_idx=site_idx,grid_path=grid_path,grid_format='ASCII')
```

The new arguments used in this code are described below:

1. The `atoms_path` and `site_idx` are the same as for *get_adsorbate()*.
2. The `grid_path` keyword argument is the path to the PEG file.
3. The `grid_format` keyword argument can be either `grid_format='cube'` or `grid_format='ASCII'`.

The result of running the previous example code is the following structure.



Advanced Skills

The previous tutorials should get you up and running so you can automate the construction of adsorbate geometries for a wide range of MOFs. However, for more advanced users, there are some additional features that may be of use. Let's walk through some more complex examples.

Adding Multiple Adsorbates

Instead of adding a single adsorbate, you may want to add an adsorbate to every metal site in a given MOF. We will consider the `Ni-BTP` MOF from the diatomic tutorial, except this time we will add an H2O molecule to every Ni site. This can be done using the code below.

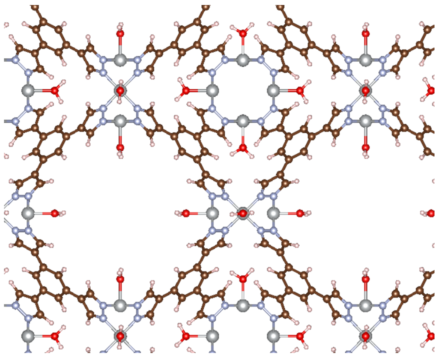
```
import os
from mai.adsorbate_constructor import adsorbate_constructor
from ase.io import read, write

starting_mof_path = os.path.join('example_MOFs','Ni-BTP.cif') #path to CIF of MOF

#Get all Ni indices in ASE Atoms object of MOF
start_mof = read(starting_mof_path)
Ni_idx = [atom.index for atom in start_mof if atom.symbol == 'Ni']

#add H2O adsorbate
atoms = start_mof
for i, site_idx in enumerate(Ni_idx):
        ads = adsorbate_constructor(ads='HOH',d_MX1=2.0,d_X1X2=0.96,d_X2X3=0.96,
                ang_MX1X2=120,ang_triads=104.5,connect=2)
        atoms = ads.get_adsorbate(atoms=atoms,site_idx=site_idx,write_file=False)
```

(continues on next page)

(continued from previous page)

```
#Write out final CIF with all H2O molecules added
if not os.path.isdir('new_mofs'):
        os.makedirs('new_mofs')
write(os.path.join('new_mofs','Ni-BTP_allH2O.cif'),atoms)
```

There are a few changes we've made to the usual workflow. The first is that we needed to identify all of the possible adsorption site indices, which we have defined as `Ni_idx`. This can be done using ASE's built-in tools for working with ASE `Atoms` ojbects. We then iterate over each site and add an H2O molecule. Since MAI adds adsorbates sequentially, we don't want to write out all the intermediate structures, so we also set `write_file=False` in *`get_adsorbate()`*. The last new aspect to introduce is that, instead of passing a CIF file via `atoms_path` in *`get_adsorbate()`*, we can directly pass in an ASE `Atoms` object via the `atoms` keyword argument. This is useful when you make modifications to the same MOF in a loop, as done here. The result of running this code is shown below.



Neighbor Algorithms

When specifying `site_idx`, the *`get_adsorbate()`* function will automatically use Pymatgen's built-in nearest neighbor algorithms to determine the atoms in the first coordination sphere of the adsorption site. By default, MAI uses Pymatgen's `crystal` algorithm, but additional algorithms are available as listed in *`get_NNs_pm()`* and described here. Through iterative testing, we have found `crystal` to be the best-performing algorithm for MOFs in general, but other algorithms can be considered if desired.

Pre-specification of Indices

In addition to the `site_idx` keyword in the *`get_adsorbate()`* function, it is also possible to manually specify the indices of the atoms in the first coordination sphere via the `NN_indices` keyword argument. This flexibility makes it possible to use a wide range of workflows in determining the desired adsorption site and coordinating atoms.

1.3 Citing MAI

If you use MAI in your work, please cite "Identifying Promising Metal-Organic Frameworks for Heterogeneous Catalysis via High-Throughput Periodic Density Functional Theory" by A.S. Rosen, J.M. Notestein, and R.Q. Snurr. If you feel so inclined, you can also reference the corresponding Zenodo DOI for the MAI code: 10.5281/zenodo.1451875.

1.4 Code documentation

1.4.1 mai.ads_sites module

class mai.ads_sites.**ads_pos_optimizer**(*adsorbate_constructor, write_file=True, new_mofs_path=None, error_path=None, log_stats=True*)

Bases: `object`

This identifies ideal adsorption sites

Args: adsorbate_constructor (class): adsorbate_constructor class containing many relevant defaults

write_file (bool): if True, the new ASE atoms object should be written to a CIF file (defaults to True)

new_mofs_path (string): path to store the new CIF files if write_file is True (defaults to /new_mofs)

error_path (string): path to store any adsorbates flagged as problematic (defaults to /errors)

log_stats (bool): print stats about process

check_and_write(*new_mof, new_name*)

Check for overlapping atoms and write CIF file

Args: new_mof (ASE Atoms object): the new MOF-adsorbate complex

new_name (string): the name of the new CIF file to write

Returns: overlap (boolean): True or False for overlapping atoms

construct_mof(*mof, ads_pos, site_idx*)

Construct the MOF-adsorbate complex

Args: ads_pos_optimizer (class): see ads_sites.py for details

ads (string): adsorbate species

ads_pos (numpy array): 1D numpy array for the proposed adsorption position

Returns: mof (ASE Atoms object): ASE Atoms object with adsorbate

get_NNs(*ads_pos, site_idx*)

Get the number of atoms nearby the proposed adsorption site within r_cut

Args: ads_pos (numpy array): 1D numpy array for the proposed adsorption position

site_idx (int): ASE index for adsorption site

Returns: NN (int): number of neighbors within r_cut

min_dist (float): distance from adsorbate to nearest atom

n_overlap (int): number of overlapping atoms

get_bi_ads_pos(*normal_vec, center_coord, site_idx*)

Get adsorption site for a 2-coordinate site

Args: normal_vec (numpy array): 1D numpy array for the normal vector to the line

center_coord (numpy array): 1D numpy array for adsorption site

site_idx (int): ASE index of adsorption site

Returns: ads_pos (numpy array): 1D numpy array for the proposed adsorption position

get_dist_planar(*normal_vec*)

Get distance vector for planar adsorption site

Args: normal_vec (numpy array): 1D numpy array for normal vector

Returns: dist (float): distance vector scaled to d_MX1

get_new_atoms (*ads_pos*, *site_idx*)
Get new ASE atoms object with adsorbate from pymatgen analysis

Args: ads_pos (numpy array): 1D numpy array for the proposed adsorption position

Returns: new_mof (ASE Atoms object): new ASE Atoms object with adsorbate

get_new_atoms_grid (*site_pos*, *ads_pos*)
Get new ASE atoms object with adsorbate from energy grid

Args: ads_pos (numpy array): 1D numpy array for the proposed adsorption position

Returns: new_mof (ASE Atoms object): new ASE Atoms object with adsorbate

get_nonplanar_ads_pos (*scaled_sum_dist*, *center_coord*)
Get adsorption site for non-planar structure

Args: scaled_sum_dist (numpy array): 2D numpy array for the scaled Euclidean distance vectors between each coordinating atom and the central atom (i.e. the adsorption site)

center_coord (numpy array): 1D numpy array for adsorption site

Returns: ads_pos (numpy array): 1D numpy array for the proposed adsorption position

get_opt_ads_pos (*mic_coords*, *site_idx*)
Get the optimal adsorption site

Args: mic_coords (numpy array): 2D numpy array for the coordinates of each coordinating atom using the central atom (i.e. adsorption site) as the origin

site_idx (int): ASE index of adsorption site

Returns: ads_pos (numpy array): 1D numpy array for the proposed adsorption position

get_planar_ads_pos (*center_coord*, *dist*, *site_idx*)
Get adsorption site for planar structure

Args: center_coord (numpy array): 1D numpy array for adsorption site (i.e. the central atom)

site_idx (int): ASE index for adsorption site

Returns: ads_pos (numpy array): 1D numpy array for the proposed adsorption position

get_tri_ads_pos (*normal_vec*, *scaled_sum_dist*, *center_coord*, *site_idx*)
Get adsorption site for a 3-coordinate site

Args: normal_vec (numpy array): 1D numpy array for the normal vector to the line

scaled_sum_dist (numpy array): 2D numpy array for the scaled Euclidean distance vectors between each coordinating atom and the central atom (i.e. the adsorption site)

center_coord (numpy array): 1D numpy array for adsorption site

site_idx (int): ASE index of adsorption site

Returns: ads_pos (numpy array): 1D numpy array for the proposed adsorption position

1.4.2 mai.adsorbate_constructor module

class mai.adsorbate_constructor.**adsorbate_constructor**(*ads='X', d_MX1=2.0, eta=1, connect=1, d_X1X2=1.25, d_X2X3=None, ang_MX1X2=None, ang_triads=None, r_cut=2.5, sum_tol=0.5, rmse_tol=0.25, overlap_tol=0.75*)

Bases: object

This class constructs an ASE atoms object with an adsorbate Initialized variables

Args: ads (string): string of element or molecule for adsorbate (defaults to 'X')

d_MX1 (float): distance between adsorbate and surface atom. If used with get_adsorbate_grid, it represents the maximum distance (defaults to 2.0)

eta (int): denticity of end-on (1) or side-on (2) (defaults to 1)

connect (int): the connecting atom in the species string (defaults to 1)

d_X1X2 (float): X1-X2 bond length (defaults to 1.25)

d_X2X3 (float): X2-X3 bond length for connect == 1 or X1-X3 bond length for connect == 2 (defaults to d_bond1)

ang_MX1X2 (float): site-X1-X2 angle (for diatomics, defaults to 180 degrees except for side-on in which it defaults to 90 or end-on O2 in which it defaults to 120; for triatomics, defaults to 180 except for H2O in which it defaults to 104.5)

ang_triads (float): X3-X1-X2 angle (defaults to 180 degrees for connect == 1 and 90 degrees for connect == 2)

r_cut (float): cutoff distance for calculating nearby atoms when ranking adsorption sites

sum_tol (float): threshold to determine planarity. when the sum of the Euclidean distance vectors of coordinating atoms is less than sum_tol, planarity is assumed

rmse_tol (float): second threshold to determine planarity. when the root mean square error of the best-fit plane is less than rmse_tol, planarity is assumed

overlap_tol (float): distance below which atoms are assumed to be overlapping

get_adsorbate(*atoms_path=None, site_idx=None, omd_path=None, NN_method='crystal', allowed_sites=None, write_file=True, new_mofs_path=None, error_path=None, NN_indices=None, atoms=None, new_atoms_name=None*)

Add an adsorbate using PymatgenNN or OMD

Args:

atoms_path (string): filepath to the CIF file

site_idx (int): ASE index for the adsorption site

omd_path (string): filepath to OMD results folder (defaults to '/oms_results')

NN_method (string): string representing the desired Pymatgen nearest neighbor algorithm. options include 'crystal',vire','okeefe', and others. See NN_algos.py (defaults to 'crystal')

allowed_sites (list of strings): list of allowed site species for use with automatic OMS detection

write_file (bool): if True, the new ASE atoms object should be written to a CIF file (defaults to True) new_mofs_path (string): path to store the new CIF files if write_file is True (defaults to /new_mofs within the directory containing the starting CIF file)

error_path (string): path to store any adsorbates flagged as problematic (defaults to /errors within the directory containing the starting CIF file)

NN_indices (list of ints): list of indices for first coordination sphere (these are usually automatically detected via the default of None)

atoms (ASE Atoms object): the ASE Atoms object of the MOF to add the adsorbate to (only include if atoms_path is not specified)

new_atoms_name (string): the name of the MOF used for file I/O purposes (defaults to the basename of atoms_path if provided)

Returns: new_atoms (Atoms object): ASE Atoms object of MOF with adsorbate

get_adsorbate_grid(*atoms_path=None*, *site_idx=None*, *grid_path=None*, *grid_format='ASCII'*, *write_file=True*, *new_mofs_path=None*, *error_path=None*, *atoms=None*, *new_atoms_name=None*)

This function adds a molecular adsorbate based on a potential energy grid

Args: atoms_path (string): filepath to the CIF file

site_idx (int): ASE index for the adsorption site

grid_path (string): path to the directory containing the PEG (defaults to /energy_grids)

grid_format (string): accepts either 'ASCII' or 'cube' and is the file format for the PEG (defaults to 'ASCII')

write_file (bool): if True, the new ASE atoms object should be written to a CIF file (defaults to True)

new_mofs_path (string): path to store the new CIF files if write_file is True (defaults to /new_mofs)

error_path (string): path to store any adsorbates flagged as problematic (defaults to /errors)

atoms (ASE Atoms object): the ASE Atoms object of the MOF to add the adsorbate to (only include if atoms_path is not specified)

new_atoms_name (string): the name of the MOF used for file I/O purposes (defaults to the basename of atoms_path if provided)

Returns: new_atoms (Atoms object): ASE Atoms object of MOF with adsorbate

1.4.3 mai.grid_handler module

mai.grid_handler.**cube_to_xyzE**(*cube_file*)

Converts cube to ASCII file Adopted from code by Julen Larrucea Original source: https://github.com/julenl/molecular_modeling_scripts

Args: cube_file (string): path to cube file

Returns: pd_data (Pandas dataframe): dataframe of (x,y,z,E) grid

mai.grid_handler.**get_best_grid_pos**(*atoms*, *max_dist*, *site_idx*, *grid_filepath*)

Finds minimum energy position in grid dataframe

Args: atoms (ASE Atoms object): Atoms object of structure

max_dist (float): maximum distance from active site to consider

site_idx (int): ASE index of adsorption site grid_filepath (string): path to energy grid

Returns: ads_pos (array): 1D numpy array for the ideal adsorption position

`mai.grid_handler.`**`grid_within_cutoff`**(*df*, *atoms*, *max_dist*, *site_pos*, *partition=1000000.0*)
Reduces grid dataframe into data within max_dist of active site

Args: df (pandas df object): df containing energy grid details (x,y,z,E)

atoms (ASE Atoms object): Atoms object of structure

max_dist (float): maximum distance from active site to consider

site_pos (array): numpy array of the adsorption site

partition (float): how many data points to partition the df for. This is used to prevent memory overflow errors. Decrease if memory errors arise.

Returns: new_df (pandas df object): modified df only around max_dist from active site and also with a new distance (d) column

`mai.grid_handler.`**`read_grid`**(*grid_filepath*)
Convert energy grid to pandas dataframe

Args: grid_filepath (string): path to energy grid (must be .cube or .grid)

Returns: df (pandas df object): df containing energy grid details (x,y,z,E)

1.4.4 mai.NN_algos module

`mai.NN_algos.`**`get_NNs_pm`**(*atoms*, *site_idx*, *NN_method*)
Get coordinating atoms to the adsorption site

Args: atoms (Atoms object): atoms object of MOF

site_idx (int): ASE index of adsorption site

NN_method (string): string representing the desired Pymatgen nearest neighbor algorithm: refer to http://pymatgen.org/_modules/pymatgen/analysis/local_env.html

Returns: neighbors_idx (list of ints): ASE indices of coordinating atoms

1.4.5 mai.oms_handler module

`mai.oms_handler.`**`get_ase_NN_idx`**(*atoms*, *coords*)
Get the ASE indices for the coordinating atoms

Args: atoms (Atoms object): ASE Atoms object for the MOF

coords (numpy array): coordinates of the coordinating atoms

Returns: ase_NN_idx (list of ints): ASE indices of the coordinating atoms

`mai.oms_handler.`**`get_ase_oms_idx`**(*atoms*, *coords*)
Get the ASE index of the OMS

Args: atoms (Atoms object): ASE Atoms object for the MOF

coords (numpy array): coordinates of the OMS

Returns: ase_oms_idx (int): ASE index of OMS

`mai.oms_handler.`**`get_omd_data`**(*oms_data_path*, *name*, *atoms*)
Get info about the open metal site from OpenMetalDetector results files

Args: oms_data_path (string): path to the OpenMetalDetector results

name (string): name of the MOF

atoms (ASE Atoms object): Atoms object for the MOF

Returns: omsex_dict (dict): dictionary of data from the OpenMetalDetector results

1.4.6 mai.regression module

`mai.regression.`**`OLS_fit`**(*xyz*)

Make ordinary least squares fit to z=a+bx+cy and return the normal vector

Args: xyz (numpy array): 2D numpy array of XYZ values (N rows, 3 cols)

Returns: normal_vec (numpy array): 1D numpy array for the normal vector

`mai.regression.`**`TLS_fit`**(*xyz*)

Make total least squares fit to ax+by+cz+d=0 and return the normal vector

Args: xyz (numpy array): 2D numpy array of XYZ values (N rows, 3 cols)

Returns: rmse (float): root mean square error of fit

normal_vec (numpy array): 1D numpy array for the normal vector

1.4.7 mai.species_rules module

`mai.species_rules.`**`add_CH4_SS`**(*mof*, *site_idx*, *ads_pos*)

Add CH4 to the structure from single-site model

Args: mof (ASE Atoms object): starting ASE Atoms object of structure

site_idx (int): ASE index of site based on single-site model

ads_pos (array): 1D numpy array for the best adsorbate position

Returns: mof (ASE Atoms object): ASE Atoms object with adsorbate

`mai.species_rules.`**`add_diatomic`**(*mof*, *ads_species*, *ads_pos*, *site_idx*, *d_X1X2=1.25*, *ang_MX1X2=None*, *eta=1*, *connect=1*, *r_cut=2.5*, *overlap_tol=0.75*)

Add diatomic to the structure

Args: mof (ASE Atoms object): starting ASE Atoms object of structure

ads_species (string): adsorbate species

ads_pos (array): 1D numpy array for the best adsorbate position

site_idx (int): ASE index of site

d_X1X2 (float): X1-X2 bond length (defaults to 1.25)

ang_MX1X2 (float): site-X1-X2 angle (defaults to 180 degrees except for side-on in which it defaults to 90 degrees prior to the centering)

eta (int): denticity of end-on (1) or side-on (2) (defaults to 1)

connect (int): the connecting atom in the species string (defaults to 1)

r_cut (float): cutoff distance for calculating nearby atoms when ranking adsorption sites (defualts to 2.5)

overlap_tol (float): distance below which atoms are assumed to be overlapping (defualts to 0.75)

Returns: mof (ASE Atoms object): ASE Atoms object with adsorbate

`mai.species_rules.add_monoatomic`(*mof*, *ads_species*, *ads_pos*)

Add adsorbate to the ASE atoms object

Args: mof (ASE Atoms object): starting ASE Atoms object of structure

ads_species (string): adsorbate species

ads_pos (numpy array): 1D numpy array for the proposed adsorption position

Returns: mof (ASE Atoms object): ASE Atoms object with adsorbate

`mai.species_rules.add_triatomic`(*mof*, *ads_species*, *ads_pos*, *site_idx*, *d_X1X2=1.25*, *d_X2X3=None*, *ang_MX1X2=None*, *ang_triads=None*, *connect=1*, *r_cut=2.5*, *overlap_tol=0.75*)

Add triatomic to the structure

Args: mof (ASE Atoms object): starting ASE Atoms object of structure

ads_species (string): adsorbate species

ads_pos (array): 1D numpy array for the best adsorbate position

site_idx (int): ASE index of site

d_X1X2 (float): X1-X2 bond length (defaults to 1.25)

d_X2X3 (float): X2-X3 bond length for connect == 1 or X1-X3 bond length for connect == 2 (defaults to d_X1X2)

ang_MX1X2 (float): site-X1-X2 angle (defaults to 180 degrees)

ang_triads (float): triatomic angle (defaults to 180 degrees for connect == 1 and ang_MX1X2 for connect == 2)

connect (int): the connecting atom in the species string (defaults to 1)

r_cut (float): cutoff distance for calculating nearby atoms when ranking adsorption sites (defualts to 2.5)

overlap_tol (float): distance below which atoms are assumed to be overlapping (defualts to 0.75)

Returns: mof (ASE Atoms object): ASE Atoms object with adsorbate

1.4.8 mai.tools module

`mai.tools.get_refcode`(*atoms_filename*)

Get the name of the MOF

Args: atoms_filename (string): filename of the ASE Atoms object (accepts CIFS, POSCARs, and CONTCARs)

Return: refcode (string): name of MOF (defaults to 'mof' if the original filename is just named CONTCAR or POSCAR)

`mai.tools.string_to_formula`(*species_string*)

Convert a species string to a chemical formula

Args: species_string (string): string of atomic/molecular species

Return: formula (string): stoichiometric chemical formula

Python Module Index

Index