CatMAP Documentation

Release 0.2.79

Andrew J. Medford

May 19, 2022

Contents

A brief overview of the purpose of this tool is provided in *Code Overview*. For more information on how to use CatMAP have a look at *Installation* and *Tutorials*. If you are interested in developing see the *Developer Info* page, and sign up for the mailing list.

CHAPTER 1

Installation

CatMAP is currently in alpha testing and thus only can only be installed from source via GitHub. The code runs directly from source, so it can be “installed” by cloning the GitHub repository. Before installing make sure that the required dependencies are installed:

- python 2.5 or greater
- numpy
- scipy
- matplotlib
- mpmath
- ase
- gmpy (optional - gives 2-3x speedup)

You can check that the dependencies are installed by starting a python session and importing them:

```
bash $ python Python >2.5 (....)
Type "help", "copyright", "credits" or "license" for more information.
>>>import mpmath
>>>import matplotlib
>>>import numpy
>>>import scipy
>>>import ase
>>>import gmpy
```

After ensuring that you have the dependencies, change to the directory where you want the CatMAP source to be installed (we will call it $CATMAP):

```
$ cd $CATMAP
$ git clone https://github.com/SUNCAT-Center/catmap.git
```

This will clone the repository into a directory called “catmap”. Next, you need to add the location of the CatMAP source code to the $PYTHONPATH environment variable so that python knows where to find it:

BASH:

```
bash $ export PYTHONPATH=$CATMAP/catmap:$PYTHONPATH
```

CSHELL:

```
$ setenv PYTHONPATH=$CATMAP/catmap:$PYTHONPATH
```

You can verify that everything went smoothly by importing the CatMAP module:

```
$ python >>>import catmap
```

Documentation (this wiki) is located in the catmap/docs folder, and is available online at http://catmap.readthedocs.org/. The best place to start learning how to use the code is the *Tutorials*.

CHAPTER 2

Tutorials

This page presents a collection of tutorials. It is suggested that first-time users follow the numbered tutorials sequentially since they are aimed at teaching basic usage of the software. Tutorials 1-3 focus on constructing a "scaled" model as a function of descriptor space, and are the core tutorials. The other tutorials explore some additional features and do not need to be followed sequentially. These will also be updated as new features are added.

2.1 Generating an Input File

The first step for any kinetic model is to properly generate an input file. Input files are processed by a "parser" class of CatMAP. The job of the parser is to convert some standard input into a Python data structure compatible with the kinetic model. The default parser is the TableParser which has been designed to accept inputs in a tabular format. This tutorial is designed to explain how users can create their own input files for any model which uses the TableParser (currently the only "parser" class implemented). If users are interested in creating inputs in some other format then it is also possible to design custom "parser" classes, but this is advanced and will not be discussed further here.

The tutorial is broken into two parts: an *overview* of input file structure and an example of how to create an input file for methane synthesis on $\mathrm{Rh}(111)$ from DFT.

2.1.1 Input File Overview

File Structure

The TableParser accepts inputs in a tab-separated text file. An example of the header and first few lines are provided below:

```
surface_name    site_name   species_name  formation_energy    bulk_structure ␣
↪frequencies reference
None            gas         CH4              0                    None          []    ␣
↪       Defined as part of reference state to have formation_energy of 0
None            gas         H2O              0                    None          []    ␣
↪       Defined as part of reference state to have formation_energy of 0
```

(continues on next page)

(continued from previous page)

```
None            gas         H2              0                   None            ␣
↪[4401]      Defined as part of reference state to have formation_energy of 0
None            gas         CO              2.74                None            ␣
↪[2170]      Energy Environ. Sci., 3, 1311-1315 (2010)
Pt              211         CO              1.113               fcc             []  ␣
↪      J. Phys. Chem. C, 113 (24), 10548-10553 (2009)
```

(Note that spaces instead of tab characters have been used here for readability. A functional input file should have fields separated by 1 tab character).

The first line provides the titles for all columns. Note that *column order does not matter* but *header names do matter*. In other words, you could put the "site_name" column before the "surface_name" column, but you could not call the "site_name" column "site_labels". The following column titles are *required for a functional input file*:

- surface_name
- site_name
- species_name
- formation_energy
- frequencies
- reference

Any number of other input columns may be added but they will be ignored by default (like the "bulk_structure" column in this example). Details on how to parse in additional data will be covered in a future tutorial.

The text input files can be generated by using LibreOffice, Excel, or other spreadsheet programs by creating a spreadsheet with the appropriate columns and exporting as a tab separated value. Alternatively they can fairly easily be generated within Python or other programming languages.

Field Values

A brief explanation of the inputs for each field is provided below. Some references are made to the ReactionModel attributes which are used to decide which fields are parsed in. If you have not yet read the other documentation some of this might not make sense at first, but after working through other examples it should be more clear.

surface_name

This is the name of the surface to which a species is adsorbed. The names are arbitrary, but only names which appear in the "surface_names" attribute of the ReactionModel will be parsed in. The only special value is "None" which will be parsed in regardless of whether or not it appears in the "surface_names" attribute. This is typically used for gas-phase species.

site_name

These names are used to distinguish different site types. They can correspond to a facet ("111" or "211") or be more specific ("hcp", "fcc" or "top"). Gas-phase species should have the site defined as "gas". Site names which appear in the { {species_definitions[site]['site_names']} } list will be parsed in, where "site" is the designation of any site in the model.

species_name

The names of the adsorbates, transition-states, and gasses are defined here. In principle an arbitrary string can be used (e.g. "methoxy") but it is often practical to use a chemical formula (e.g. CH3O) since the composition of such strings can be automatically determined (the ASE function ase.atoms.strings2symbols is used). If the name cannot be parsed by ase.atoms.strings2symbols then the atomic composition must be manually specified in the {species_definitions[species_name]['composition']} dictionary. For example:

```
species_definitions['methoxy']['composition'] = {'C':1,'H':3,'O':1}
```

Only species which appear in the "species_names", "gas_names", and "transition_state_names" attributes of the ReactionModel will be parsed. These attributes are typically specified automatically by the "rxn_expressions" strings, so if, for some reason, you want to parse in a species which is not in the reaction network you can do so by adding a "dummy" reaction like "CO_g -> CO_g" to have the model parse in the energetics of gas-phase CO.

formation_energy

This is the core of the input file since it defines the energetics of the system. It should be a "relative free energy of formation" (see *Formation Energy Approach*) of the "species_name" on the "surface_name" and "site_name". A "relative free energy of formation" is different from a standard free energy of formation in that a "standard free energy of formation" uses pure elements as the reference states. In contrast, a "relative free energy of formation" can include heteroatomic molecules (such as CO) within the reference state, such that the relative energies of molecules in the reference state are set equal to 0. For species on surfaces, formation energies are often not available experimentally, and thus there is great value in computing them by an electronic structure method such as DFT. For gas phase species, experimental values are more accurate and can be used directly or as a correction for DFT values. To perform thermodynamic calculations, it is generally necessary that all energies share a common thermodynamic reference state for the species in the reaction mechanism. (see *Formation Energy Approach*).

frequencies

This is a list of the vibrational frequencies of the "species_name" on the "surface_name" at the "site_name". Although this field is required, it is possible to input an empty list "[]" if the vibrational frequencies are not known. The vibrational frequencies are used to compute the zero-point and free energy corrections for gas phase and adsorbed species. By default the units are assumed to be "wavenumbers" or "cm^-1", but this can be changed by editing the "frequency_unit_conversion" variable (1.239842e-4 by default) so that input_frequency*frequency_unit_conversion = input_frequency [eV]. Gas-phase vibrational frequencies can be found in NIST (be careful since redundant frequencies are listed only once) and some are compiled in the catmap.data.experimental_gas_frequencies dictionary. Vibrational frequencies of adsorbed species can be costly to compute, and hence a few approximations are sometimes employed. These approximations are controlled by the "estimate_frequencies" attribute of the TableParser. The values, in order of increasing accuracy, are:

- estimate_frequencies >3: Use empty frequency set for species without any frequencies specified.
- estimate_frequencies >= 3: Use frequencies of atomic species (e.g. $\nu_{CH_4} = \nu_C + 4 * \nu_H$ where ν_X is a Python list of the vibrational species of species X adsorbed)
- estimate_frequencies >= 2: Estimate frequency of transition-states from the dissociated state frequency (e.g. $\nu_{C-O} = \nu_C + \nu_O$)
- estimate_frequencies >= 1: Estimate frequency of adsorbed state at one site using frequency from other sites (e.g. $\nu_{CO(111)} = \nu_{CO(211)}$)
- estimate_frequencies = 0: Only accept frequencies from the exact adsorbate on the correct site. However, a single set of frequencies will still be used for all surfaces. If the attribute "frequency_surface_names" is

defined then an average of the frequencies from the surface(s) in this list will be used. Otherwise an average of all available frequencies for each adsorbate will be used. For example, to use only Cu vibrational frequencies set {{frequency_surface_names = ['Cu']}}, or to average Cu and Pt vibrational frequencies use {{frequency_surface_names = ['Cu', 'Pt']}}. Allowing frequencies to vary with site would require a way of estimating frequency as a function of descriptors and is not currently implemented.

reference

This is an arbitrary string which notes the source of the information. Usually a publication/citation is provided for previously computed work, or for your own input you could use "Unpublished", "This work", "DFT/GPAW/RPBE", etc. This is used when generating a summary file for the model, and it is always good practice to note the source of inputs.

Formation Energy Approach

One key point for generating input files is that the energies are computed as a "relative free energies of formation" relative to a *common reference* state. This relative free energy of formation is provided by:

$G_i = H_i - T * S_i - \sum_j (n_j R_j)$

G_i is the "relative Gibbs free energy of formation" of species i . H_i is the enthalpy of species i (see further below). S_i is the absolute entropy of species i . nj is the number of atomic species j in i, and $|R_j|$ is the reference Gibbs free energy of that atomic species. Mathematically this looks a little confusing (especially with such crude notation) but in practice it is pretty easy. The general principle is similar to https://en.wikipedia.org/wiki/Born%E2%80%93Haber_cycle and https://en.wikipedia.org/wiki/Hess%27s_law For this type of input file of CatMAP, we will be specifying the 0K electronic contribution to the relative formation energies, and these do not have a temperature depndence. How other terms become included will be explained later.

In practice, today, the value for H_i is generally approximated as being equal to the electronic energy. In this case, the equation becomes

$G_i = U_i - T * S_i - \sum_j (n_j R_j)$

Where U_i is the raw/DFT energy of species i,

Here, we will provide a simple example of how to provide the electronic energies information that CatMAP needs to calculate this contribution for the relative free energy of formation. We will use the variable E_i to emphasize that this term, based on *U_i*, represents the electronic energy contribution for G_i .

In CatMAP, the the Zero Point Energy (ZPE) correction terms should not be included in *U_i*, The ZPE correction terms for *U_i* will be added later by CatMAP based on the vibrational frequencies provided. The $-T * S_i$ term will also be added later by CatMAP

Let's look at an example. Say we want to find the energy of gas-phase CO relative to carbon (C) in methane (CH_4), oxygen (O) in H_2O, and hydrogen (H) in molecular hydrogen (H_2). We first compute the reference energies ($|R_j|$) for each atomic species:

$$R_\mathrm{H} = 0.5(U_{\mathrm{H}_2})$$
$$R_\mathrm{C} = U_{\mathrm{CH}_4} - 4R_\mathrm{H}$$
$$R_\mathrm{O} = U_{\mathrm{H}_2\mathrm{O}} - 2R_\mathrm{H}$$

(where again U is a "raw" energy from an ab-initio calculation, or a "regular" formation energy from NIST).

Now we can compute the electronic contribution to the relative formation energy of CO as:

$E_\mathrm{CO} = U_\mathrm{CO} - R_\mathrm{C} - R_\mathrm{O}$

In the case where CO is adsorbed to a surface, say Pt(211), we can compute the electronic contribution to the relative formation energy relative to the clean surface:

$E_{\mathrm{CO*@Pt(211)}} = U_{\mathrm{Pt(211)+CO*}} - U_{\mathrm{Pt(211)}} - R_{\mathrm{C}} - R_{\mathrm{O}}$

One nice thing about the formation energy approach is that it does not distinguish between thermodynamic minima (adsorbed states) and saddle points (transition-states). Thus, it is possible to compute a formation energy of the $\mathrm{C-O}$ dissociation transition-state on $\mathrm{Pt(211)}$ as:

$E_{\mathrm{C-O@Pt(211)}} = U_{\mathrm{Pt(211)+C-O}} - U_{\mathrm{Pt(211)}} - R_{\mathrm{C}} - R_{\mathrm{O}}$

Then one could compute the barrier for $\mathrm{C-O}$ dissociation as:

$E_{\mathrm{C-O@Pt(211)}} - E_{\mathrm{CO*@Pt(211)}}$

If this still doesn't make sense, try working through the *example* below.

In principle the choice of reference states is arbitrary since the reference energies $|R_j|$ cancel out in any relative quantities. However, in many cases it is necessary to use some correction scheme for the gas-phase energies if they are poorly described by the level of theory used (e.g. DFT). In this case it is best to select a reference set for which the reference species are well-described by the level of theory. For example, it is well-known that $\mathrm{O_2}$ and $\mathrm{CO_2}$ are not properly described by DFT, so it would not make sense to use these to compute the reference energies $|R_j|$.

It is also worth re-iterating that the *same reference energies* $|R_j|$ *must be used for all energies in a given input file*. The best practice is to first set any pure element reactants in the system as having relative free energies of formation of 0 and then to add in gases with one additional element as having relative free energies of formation of 0. Finally, other species (the remaining species with elements already used) will have relative free energies of formation defined based on these reference states as well as the math:*U_i* (or math:*U_i - T*S_i*) computed values for these other / remaining species. When looking at an input file that has been created correctly, the gas-phase species that were used as part of the reference state are easy to recognize since their relative formation energies will be set to 0. (see *overview*).

Formation Energy Approach and Temperature Dependence

The below paragraphs are for informational purposes only.

As noted above, a free energy of formation has several terms:

$G_i = H_i - T * S_i - \sum_j (n_j R_j)$

In general, for formation energies, the Temperature and Pressure and any other quantities used for defining the reference states should be reported in the manuscript (whether using a relative formation energy or a standard formation energy). The temperature dependence and entropy contributions are handled elsewhere in CatMAP, and are thus not included in the "formation_energy" field of the input file.

The best practice and state of the art today is to include the entropy of formation, S_i when calculating G_i The value from $T * S_i$ (and the values within $|R_j|$) will then include the values for the entropy contribtuions calculated at a given temperature based on the partition functions for vibrations, rotations, and the Sackur-Tetrode equation. The Sackur-Tetrode equation includes both the translational partition function contribution and a quantum configurational term. (The Sackur-Tetrode equation is often referred to as simply the "translational entropy", which can be misleading).

Computational calculation of the entropy contribution to G_i has a significant computational expense (because it requires more than single point calculations), and many studies do not require that level of accuracy since for many systems changes in U_i affect the chemistry and kinetics more than changes in S_i

When the term $T * S_i$ is approximated as sufficiently insignificant, the equation reduces to:

$G_i = U_i - \sum_j (n_j R_j)$

However, CatMAP supports an inclusion of an approximate entropy, and use of this feature is encouraged. CatMAP uses existing codes to calculate the entropy contributions of for vibrational stretching modes and gas phase translations from statistical mechanics, and thus we include stretching mode vibrational frequencies near the end of the example below.

CatMAP's built-in thermo corrections will then use the frequencies to add in ZPE + enthalpy + entropy to complete *G_i*. In the future, even more accurate entropy terms may be included, but present day studies are adequately served by using stretching mode contributions for adsorbates.

2.1.2 Example

In this example we will generate an input file for methane synthesis from CO and $\mathrm{H_2}$ (methanation) on Rh(111) using some previously computed DFT values and a Python script. You can copy-paste the code as you go along, or find the whole script at GitHub.

Take the simplified methanation reaction mechanism:

- $\mathrm{CO_{gas}} + * \rightarrow \mathrm{CO}*$
- $\mathrm{CO} * + * \rightarrow \mathrm{C} * + \mathrm{O}*$
- $\mathrm{O} * + \mathrm{H}* \leftrightarrow \mathrm{OH}*$ (quasi-equilibrated)
- $\mathrm{OH} * + \mathrm{H}* \rightarrow \mathrm{H_2O_{gas}} + 2*$
- $\mathrm{C} * + \mathrm{H}* \rightarrow \mathrm{CH} * + *$
- $\mathrm{CH} * + \mathrm{H}* \leftrightarrow \mathrm{CH_2} * + *$ (quasi-equilibrated)
- $\mathrm{CH_2} * + \mathrm{H}* \leftrightarrow \mathrm{CH_3} * + *$ (quasi-equilibrated)
- $\mathrm{CH_3} * + \mathrm{H}* \leftrightarrow \mathrm{CH_{4,gas}} + 2*$ (quasi-equilibrated)

Where * denotes a Rh(111) site. For this example we need energies of the following species:

- CO (gas)
- $\mathrm{H_2}$ (gas)
- $\mathrm{CH_4}$ (gas)
- $\mathrm{H_2O}$ (gas)
- CO (adsorbed)
- O (adsorbed)
- C (adsorbed)
- H (adsorbed)
- CH (adsorbed)
- OH (adsorbed)
- $\mathrm{CH_2}$ (adsorbed)
- $\mathrm{CH_3}$ (adsorbed)
- $\mathrm{C-O}$ (transition-state)
- $\mathrm{H-OH}$ (transition-state)
- $\mathrm{H-C}$ (transition-state)
- (111 slab)

Let's assume that we have computed the energies of these species on a Rh(111) surface using some ab-initio method and stored them in a Python dictionary:

```
abinitio_energies = {
         'CO_gas': -626.611970497,
         'H2_gas': -32.9625308725,
         'CH4_gas': -231.60983421,
         'H2O_gas': -496.411394229,
         'CO_111': -115390.445596,
         'C_111': -114926.212205,
         'O_111': -115225.106527,
         'H_111': -114779.038569,
         'CH_111': -114943.455431,
         'OH_111': -115241.861661,
         'CH2_111': -114959.776961,
         'CH3_111': -114976.7397,
         'C-O_111': -115386.76440668429,
         'H-OH_111': -115257.78796158083,
         'H-C_111': -114942.25042955727,
         'slab_111': -114762.254842,
         }
```

(in this case the energies were generated by Quantum Espresso)

Next, we need to decide on a choice of reference molecules. One simple option for this system is to take hydrogen relative to H_2, carbon relative to CH_4, and oxygen relative to H_2O. We will take all adsorption energies relative to the clean (111) Rh slab.

```
ref_dict = {}
ref_dict['H'] = 0.5*abinitio_energies['H2_gas']
ref_dict['O'] = abinitio_energies['H2O_gas'] - 2*ref_dict['H']
ref_dict['C'] = abinitio_energies['CH4_gas'] - 4*ref_dict['H']
ref_dict['111'] = abinitio_energies['slab_111']
```

Now we can write a function to convert these "raw" energies to "reference" energies. Note that we use the function *ase.atoms.string2symbols* as a convenient way to get the composition from the chemical formula.

```
from ase.atoms import string2symbols

def get_formation_energies(energy_dict,ref_dict):
    formation_energies = {}
    for key in energy_dict.keys(): #iterate through keys
        E0 = energy_dict[key] #raw energy
        name,site = key.split('_') #split key into name/site
        if 'slab' not in name: #do not include empty site energy (0)
            if site == '111':
                E0 -= ref_dict[site] #subtract slab energy if adsorbed
            #remove - from transition-states
            formula = name.replace('-','')
            #get the composition as a list of atomic species
            composition = string2symbols(formula)
            #for each atomic species, subtract off the reference energy
            for atom in composition:
                E0 -= ref_dict[atom]
            #round to 3 decimals since this is the accuracy of DFT
            E0 = round(E0,3)
            formation_energies[key] = E0
    return formation_energies
```

We can check that the formation energies are reasonable (i.e. of order 1 eV):

```
formation_energies = get_formation_energies(abinitio_energies,ref_dict)

for key in formation_energies:
    print key, formation_energies[key]

>>
>> OH_111 0.323
>> H_111 -0.302
>> C_111 1.727
>> H2O_gas 0.0
>> CH_111 0.965
>> CO_111 0.943
>> H2_gas 0.0
>> C-O_111 4.624
>> CO_gas 2.522
>> O_111 0.597
>> CH3_111 0.644
>> CH4_gas 0.0
>> CH2_111 1.125
>> H-OH_111 0.878
>> H-C_111 2.17
>>
```

This looks pretty good. The energies of our reference species ($H_{2,gas}$, $CH_{4,gas}$, and $H2O_{gas}$) are all 0 as expected, and all the numbers are of order 1. Usually if something goes wrong then the numbers will be similar to the raw DFT numbers (i.e. > 100 eV). We can also compute the CO dissociation barrier as $E_{C-O} - E_{CO} = 3.68\,eV$. This is pretty high, but the surface is a close-packed (111) facet so this is not too surprising.

Before making an input file we will want to get some vibrational frequencies. Again, lets just assume that these have previously been calculated by DFT and are stored in a Python dictionary as:

```
frequency_dict = {
                'CO_gas': [2170],
                'H2_gas': [4401],
                'CH4_gas':[2917,1534,1534,3019,3019,3019,1306,
                           1306,1306],
                'H2O_gas': [3657, 1595, 3756],
                'CO_111': [60.8, 230.9, 256.0, 302.9, 469.9, 1747.3],
                'C_111': [464.9, 490.0, 535.9],
                'O_111': [359.5, 393.3, 507.0],
                'H_111': [462.8, 715.9, 982.5],
                'CH_111': [413.3, 437.5, 487.6, 709.6, 735.1, 3045.0],
                'OH_111': [55, 340.9, 396.1, 670.3, 718.0, 3681.7],
                'CH2_111': [55, 305.5, 381.3, 468.0, 663.4, 790.2, 1356.1,
                            2737.7, 3003.9],
                'CH3_111': [55, 113.5, 167.4, 621.8, 686.0, 702.5, 1381.3,
                            1417.5, 1575.8, 3026.6, 3093.2, 3098.9],
                'C-O_111': [],
                'H-OH_111': [],
                'H-C_111': []
                }
```

Now we just need a function which will put everything together into a tab-separated table with the appropriate headers. The following Python function will do this for us:

```
def make_input_file(file_name,energy_dict,frequency_dict):
```

(continues on next page)

(continued from previous page)

```
    #create a header
    header = '\t'.join(['surface_name','site_name',
                        'species_name','formation_energy',
                        'frequencies','reference'])

    lines = [] #list of lines in the output
    for key in energy_dict.keys(): #iterate through keys
        E = energy_dict[key] #raw energy
        name,site = key.split('_') #split key into name/site
        if 'slab' not in name: #do not include empty site energy (0)
            frequency = frequency_dict[key]
            if site == 'gas':
                surface = None
            else:
                surface = 'Rh'
            outline = [surface,site,name,E,frequency,'Input File Tutorial.']
            line = '\t'.join([str(w) for w in outline])
            lines.append(line)

    lines.sort() #The file is easier to read if sorted (optional)
    lines = [header] + lines #add header to top
    input_file = '\n'.join(lines) #Join the lines with a line break

    input = open(file_name,'w') #open the file name in write mode
    input.write(input_file) #write the text
    input.close() #close the file

    print 'Successfully created input file'
```

Now use this function to create the text file - in this case we call it “energies.txt”:

```
file_name = 'energies.txt'
make_input_file(file_name,formation_energies,frequency_dict)

>> Successfully created input file
```

You can view the input in a human-readable format by opening energies.txt:

```
surface_name    site_name   species_name    formation_energy    frequencies reference
None    gas CH4 0.0 [2917, 1534, 1534, 3019, 3019, 3019, 1306, 1306, 1306]  Input␣
↪File Tutorial.
None    gas CO  2.522   [2170]  Input File Tutorial.
None    gas H2  0.0 [4401]  Input File Tutorial.
None    gas H2O 0.0 [3657, 1595, 3756]  Input File Tutorial.
Rh  111 C   1.727   [464.9, 490.0, 535.9]   Input File Tutorial.
Rh  111 C-O 4.624   []  Input File Tutorial.
Rh  111 CH  0.965   [413.3, 437.5, 487.6, 709.6, 735.1, 3045.0] Input File Tutorial.
Rh  111 CH2 1.125   [55, 305.5, 381.3, 468.0, 663.4, 790.2, 1356.1, 2737.7, 3003.9]␣
↪Input File Tutorial.
Rh  111 CH3 0.644   [55, 113.5, 167.4, 621.8, 686.0, 702.5, 1381.3, 1417.5, 1575.8,␣
↪3026.6, 3093.2, 3098.9] Input File Tutorial.
Rh  111 CO  0.943   [60.8, 230.9, 256.0, 302.9, 469.9, 1747.3]  Input File Tutorial.
Rh  111 H   -0.302  [462.8, 715.9, 982.5]   Input File Tutorial.
Rh  111 H-C 2.17    []  Input File Tutorial.
Rh  111 H-OH    0.878   []  Input File Tutorial.
Rh  111 O   0.597   [359.5, 393.3, 507.0]   Input File Tutorial.
Rh  111 OH  0.323   [55, 340.9, 396.1, 670.3, 718.0, 3681.7]    Input File Tutorial.
```

This particular example only creates input for a single surface, but it is fairly easy to see how one could construct a for-loop over several surfaces to create an input file with the energetics for multiple surfaces. Alternatively if you keep your data stored in a spreadsheet it should be possible to convert everything to a common reference and export the spreadsheet as tab-separated values (remember to get the header names right!).

In case we want to check that the input can be parsed correctly, we could create a "dummy" ReactionModel and ask it to parse everything in. Normally this won't be necessary since you will have an actual ReactionModel that you want to use to test the parser (see the *Creating a Microkinetic Model* tutorial), but it is included here for reference.

```
#Test that input is parsed correctly
from catmap.model import ReactionModel
from catmap.parsers import TableParser
rxm = ReactionModel()
#The following lines are normally assigned by the setup_file
#and are thus not usually necessary.
rxm.surface_names = ['Rh']
rxm.adsorbate_names = ['CO','C','O','H','CH','OH','CH2','CH3']
rxm.transition_state_names = ['C-O','H-OH','H-C']
rxm.gas_names = ['CO_g','H2_g','CH4_g','H2O_g']
rxm.species_definitions = {'s':{'site_names':['111']}}
#Now we initialize a parser instance (also normally done by setup_file)
parser = TableParser(rxm)
parser.input_file = file_name
parser.parse()
#All structured data is stored in species_definitions; thus we can
#check that the parsing was successful by ensuring that all the
#data in the input file was collected in this dictionary.
for key in rxm.species_definitions:
    print key, rxm.species_definitions[key]
```

The output of this should contain all species in the model along with their energies, frequencies, etc.

2.2 Creating a Microkinetic Model

This tutorial provides a walk-through of how to create a very basic micro-kinetic model for CO oxidation on transition-metal (111) surfaces. More advanced features are explored in *Refining a Microkinetic Model* and the *Topics* section.

2.2.1 Setting up the model

All micro-kinetic models require a minimum of 2 files: the "setup file" and the "submission script". In addition it is almost always necessary to specify an "input file" which is used by the "parser" to extend the "setup file" (see *Generating an Input File*).

Input File

We will begin by assuming that the input file has already been generated similar to *Generating an Input File*. In fact these energies were taken from CatApp and compiled into a format compatible with CatMAP:

```
surface_name    site_name   species_name    formation_energy    bulk_structure ␣
↪frequencies other_parameters    reference
None    gas CO2 2.45    None    [1333,2349,667,667] []  "Angew. Chem. Int. Ed., 47,␣
↪4835 (2008)"
None    gas CO  2.74    None    [2170]  []  "Energy Environ. Sci., 3, 1311-1315 (2010)
↪"
```

(continues on next page)

(continued from previous page)

```
None     gas O2  5.42    None    [1580]  []  "Falsig et al (2012)"
Ru  111 O   -0.07    fcc []  []  "Falsig et al (2012)"
Ni  111 O   0.35     fcc []  []  "Falsig et al (2012)"
Rh  111 O   0.55     fcc []  []  "Falsig et al (2012)"
Cu  111 O   1.07     fcc []  []  "Falsig et al (2012)"
Pd  111 O   1.55     fcc []  []  "Falsig et al (2012)"
Pt  111 O   1.62     fcc []  []  "Falsig et al (2012)"
Ag  111 O   2.05     fcc []  []  "Falsig et al (2012)"
Au  111 O   2.61     fcc []  []  "Falsig et al (2012)"
Ru  111 CO  1.3 fcc []  []  "Angew. Chem. Int. Ed., 47, 4835 (2008)"
Rh  111 CO  1.34     fcc []  []  "Angew. Chem. Int. Ed., 47, 4835 (2008)"
Pd  111 CO  1.55     fcc []  []  "Angew. Chem. Int. Ed., 47, 4835 (2008)"
Ni  111 CO  1.63     fcc []  []  "Angew. Chem. Int. Ed., 47, 4835 (2008)"
Pt  111 CO  1.7 fcc []  []  "Angew. Chem. Int. Ed., 47, 4835 (2008)"
Cu  111 CO  2.58     fcc []  []  "Angew. Chem. Int. Ed., 47, 4835 (2008)"
Ag  111 CO  2.99     fcc []  []  "Angew. Chem. Int. Ed., 47, 4835 (2008)"
Au  111 CO  3.04     fcc []  []  "Angew. Chem. Int. Ed., 47, 4835 (2008)"
Ru  111 O-CO    2.53     fcc []  []  "Angew. Chem. Int. Ed., 47, 4835 (2008)"
Rh  111 O-CO    3.1 fcc []  []  "Angew. Chem. Int. Ed., 47, 4835 (2008)"
Ni  111 O-CO    3.25     fcc []  []  "Angew. Chem. Int. Ed., 47, 4835 (2008)"
Pt  111 O-CO    4.04     fcc []  []  "Angew. Chem. Int. Ed., 47, 4835 (2008)"
Cu  111 O-CO    4.18     fcc []  []  "Angew. Chem. Int. Ed., 47, 4835 (2008)"
Pd  111 O-CO    4.2 fcc []  []  "Angew. Chem. Int. Ed., 47, 4835 (2008)"
Ag  111 O-CO    5.05     fcc []  []  "Angew. Chem. Int. Ed., 47, 4835 (2008)"
Au  111 O-CO    5.74     fcc []  []  "Angew. Chem. Int. Ed., 47, 4835 (2008)"
Ag  111 O-O 5.98     fcc []  []  "Falsig et al (2012)"
Au  111 O-O 7.22     fcc []  []  "Falsig et al (2012)"
Cu  111 O-O 4.74     fcc []  []  "Falsig et al (2012)"
Pt  111 O-O 5.35     fcc []  []  "Falsig et al (2012)"
Rh  111 O-O 3.79     fcc []  []  "Falsig et al (2012)"
Ru  111 O-O 3.34     fcc []  []  "Falsig et al (2012)"
Pd  111 O-O 5.34     fcc []  []  "Falsig et al (2012)"
```

For this example we will name this file “energies.txt” and place it in the same directory as the other files.

Setup File

Next we will create the “setup file”. Lets make a text file named “CO_oxidation.mkm”. The suffix “.mkm” is often used to designate a micro-kinetics module setup file, but it is not required.

One of the most important aspects of the “setup file” is the “rxn_expressions” variable which defines the elementary steps in the model. For this simplified CO oxidation model we will specify these as:

```
rxn_expressions = [

               '*_s + CO_g -> CO*',
               '2*_s + O2_g <-> O-O* + *_s -> 2O*',
               'CO* +  O* <-> O-CO* + * -> CO2_g + 2*',

                   ]
```

The first expression includes CO adsorption without any activation barrier. The second includes an activated dissociative chemisorption of the oxygen molecule, and the final is an activated associative desorption of CO2. More complex models for CO oxidation could be imagined, but these elementary steps capture the key features. Note that we have only included “*” and “*_s” sites since this is a single-site model for CO oxidation. This means that all intermediates will be adsorbed at a site designated as "s". These reaction expressions will be parsed automatically in order to define the adsorbates, transition-states, gasses, and surface sites in the model.

One important thing to note is that uses some subset of gas phase energies present in your input file to generate a complete set of reference energies for every element present in your reactions. However, it can only use gas species present in your reaction network. If you'd like CatMAP to use a gas species that does not appear in your reaction network as an atomic reference, you may need to add a dummy reaction like "`H2O_g -> H2O_g`" (in the case of adding H2O gas) to "`rxn_expressions`".

Next, we need to tell the model which surfaces we are interested in.

```
surface_names = ['Pt', 'Ag', 'Cu','Rh','Pd','Au','Ru','Ni']
#surfaces to include in scaling (need to have descriptors defined for each)
```

Now we will tell the model which energies to use as descriptors:

```
descriptor_names= ['O_s','CO_s'] #descriptor names
```

The model also needs to know the ranges over which to check the descriptors, and the resolution with which to discretize this range. It is generally good to use a range which includes all metals of interest, but doesn't go too far beyond. For this example we will use a relatively low resolution (15) in order to save time.

```
descriptor_ranges = [[-1,3],[-0.5,4]]

resolution = 15
```

This means that the model will be solved for each of 15 oxygen adsorption energies between -1 and 3, for each of 15 CO adsorption energies between -0.5 and 4 (a total of 225 points in descriptor space).

Next, we set the temperature of the model (in Kelvin):

```
temperature = 500
```

In the next part we will create and explicitly set some variables in the "`species_definitions`" dictionary. This dictionary is the central place where all species-specific information is stored, but for the most part it will be populated by the "parser". However, there are a few things that need to be set explicitly. First, the gas pressures:

```
species_definitions = {}
species_definitions['CO_g'] = {'pressure':1.} #define the gas pressures
species_definitions['O2_g'] = {'pressure':1./3.}
species_definitions['CO2_g'] = {'pressure':0}
```

Next, we need to include some information about the surface site:

```
species_definitions['s'] = {'site_names': ['111'], 'total':1} #define the sites
```

This line tells the code that anything with "`111`" in the "`site_name`" column of the input file has energetics associated with an "s" site. This is a list because we might want to include multiple site_names as a single site type; for example, if we designated some sites as "`fcc`" and some as "`ontop`", but both were on the (111) surface we might instead use: "`site_name:['fcc','ontop']`".

We also need to tell the model where to store the output. By default it will create a data.pkl file which contains all the large outputs (those which would take more than 100 lines to represent with text). Lets make it store things in `CO_oxidation.pkl` instead.

```
data_file = 'CO_oxidation.pkl'
```

This concludes the attributes which need to be set for the ReactionModel itself; however, we probably want to specify a few more settings of the "parser", "scaler", "solver", and "mapper".

For convenience, all variables are specified in the same file and same format. Since we did not specify a parser, the default parser (TableParser) will be used. This could be explicitly specified with parser = 'TableParser' but this is not necessary. First we will tell the parser where to find the input table that we saved earlier:

```
input_file = 'energies.txt'
```

Next, we need to tell the model how to add free energy corrections. For this example we will use the Shomate equation for the gas thermochemistry, and assume that the adsorbates have no free energy contributions (since we don't have frequencies for them).

```
gas_thermo_mode = "shomate_gas"
adsorbate_thermo_mode = "frozen_adsorbate"
```

There are a number of other approximations built into the model. For example, gas-phase thermochemistry can be approximated by:

- `ideal_gas` - Ideal gas approximation (assumes that atoms are in ase.structure.molecule and that arguments for ase.thermochemistry.IdealGasThermo are specified in catmap.data.ideal_gas_params and that frequencies are provided)
- `shomate_gas` - Uses Shomate equation (assumes that Shomate parameters are defined in catmap.data.shomate_params)
- `fixed_entropy_gas` - Includes zero-point energy and a static entropy correction (assumes that frequencies are provided and that gas entropy is provided in catmap.data.fixed_entropy_dict (if not 0.002 eV/K is used))
- `frozen_fixed_entropy_gas` - Same as `fixed_entropy_gas` except that zero-point energy is neglected.
- `zero_point_gas` - Only includes zero-point energies and neglects entropy (assumes that frequencies are provided)
- `frozen_gas` - Do not include any corrections.

Similarly, adsorbate thermochemistry can be approximated by:

- `harmonic_adsorbate` - Use the harmonic approximation and assume all degrees of freedom are vibrational (implemented via ase.thermochemistry.HarmonicThermo and assumes that frequencies are defined)
- `hindered_adsorbate` - Use a hindered translator and hindered rotor model and assume all but three degrees of freedom are vibrational, two come from hindered translations, and one comes from a hindered rotation (implemented via ase.thermochemistry.HinderedThermo and assumes that frequencies are defined)
- `zero_point_adsorbate` - Only includes zero-point energies (assumes frequencies are defined)
- `frozen_adsorbate` - Do not include any corrections.

The next thing we want to specify are some parameters for the scaler. Since we have not explicitly specified a scaler the default *GeneralizedLinearScaler* will be used. This scaler uses a coefficient matrix to map descriptor-space to parameter space and will be discussed in more detail in a future tutorial. By default a numerical fit will be made which minimizes the error by solving an over-constrained least-squares problem in order to map the lower-dimensional "parameter space" to the higher dimensional "descriptor space". However, this fit is often unstable since fits are sometimes constructed with limited input data. In order to reduce this instability we want to place constraints on the coefficients so that adsorbates only scale with certain descriptors, and we can also force coefficients to be positive, negative, equal to a value, or in between certain values. We also need to tell the scaler how to determine transition-state energies. In this example we do this by:

```
scaling_constraint_dict = {
                           'O_s':['+',0,None],
                           'CO_s':[0,'+',None],
```

(continues on next page)

(continued from previous page)

```
                                        'O-CO_s':'initial_state',
                                        'O-O_s':'final_state',
                                        }
```

(note that the keys here include the adsorbate name and the site label separated by an underscore _) This means that for oxygen we force a positive ('+') slope for descriptor 1 (oxygen binding), a slope of 0 for descriptor 2 (CO binding), and we put no constraints on the constant. This is equivalent to saying:

$E_O = a * E_O + c$

where a must be positive. Of course in this example its trivial to see that a should be 1 and c should be 0 since of course $E_O = E_O$. We could specify this explicitly using `'O_s':[1,0,0]`. We could also impose other constraints:

- `'O_s':['-',0,None]` would force a to be negative
- `'O_s':['0:3',0,None]` would force a to be between 0 and 3
- `'O_s':[None,0,None]` would put no constraints on a
- `'O_s':[None,None,None]` would let $E_O = a * E_O + b * E_{CO} + c$ with no constraints on a, b, or c

and so on. By default the constraints would be `['+','+',None]`. In this case the algorithm will find the correct solution of $a = 1$, $c = 0$ even if the solution is unconstrained, but the constraints are still specified to provide an example. We use similar logic for the CO constraint since we know that it should depend on CO binding but not on O binding.

We also need to tell the model how to handle the transition-state scaling. We have three options:

- $E_{TS} = m * E_{IS} + n$ (`initial_state`)
- $E_{TS} = m * E_{FS} + n$ (`final_state`)
- $E_{TS} = m * \Delta E + n$ (`BEP`)

where E_{TS} is the transition-state formation energy, E_{IS} is the intitial-state (reactant) energy, E_{FS} is the final-state (product) energy for the elementary step, and ΔE is the reaction energy of the elementary step. By default `initial_state` is used, but for some elementary steps this might not make sense. The dissociative adsorption of oxygen is a great example, since the initial state energy is equal to the gas-phase energy of the oxygen molecule and is a constant. Thus, if we assumed `initial_state` scaling then we would be assuming a constant activation energy which would obviously not capture trends across surfaces. Instead, we scale with the '`final_state`'.

By default the coefficients *m* and *n* are computed by a least-squares fit. They can be accessed by the "transition_state_scaling_coefficients" attribute of the ReactionModel after the model has been run. In some cases it may be necessary to specify these coefficients manually because, for example, the transition-state energies have not been calculated. This can be achieved by using the values: 'initial_state:[*m*,*n*]' or `initial_state:[m]` where '`initial_state`' could also be '`final_state`' or '`BEP`'. If only *m* is specified then *n* will be determined by a least-squared fit. It is worth noting here that while *m* is independent of the reference used to compute the "generalized formation energies" in the input file (see *Formation Energy Approach*), *n* will depend on the references for '`initial_state`' or '`final_state`' scaling. Thus if you are using transition-state scaling values from some previously published work it is critical that the same reference sets be used.

Now we need to set some parameters that will be used by the "solver". By default the SteadyStateSolver is used. First, we tell the solver how many decimals of precision we want to use:

```
decimal_precision = 100 #precision of numbers involved
```

While 100 digits of precision seems like overkill (and it actually is here), it is often necessary to go above 50 digits due to the extreme stiffness of the reaction expressions. Using 100 digits is a good rule of thumb, and doesn't slow things down too much (especially if you have gmpy installed).

Next, we set the tolerance of the steady-state solutions:

```
tolerance = 1e-50 #all d_theta/d_t's must be less than this at the solution
```

The tolerance is the maximum allowed rate of change of surface species coverages at the steady-state solution. This should be less than the smallest rate you are interested in for the problem (i.e. the lower bound of the rate "volcano plot") but should be well above the machine epsilon at the given decimal_precision (ca. 1e-100 in this case).

Finally, we set some practical variables controlling the number of iterations allowed by the solver:

```
max_rootfinding_iterations = 100

max_bisections = 3
```

The maximum rootfinding iterations controls the number of times Newton's method iterations can be applied in the rootfinding algorithm, while the maximum bisections tells the number of times the mapper can bisect descriptor space when trying to move from one point to another. Note that the maximum number of intermediate points between two points in descriptor space is $2^{\text{max\_bisections}}$ so increasing this number can slow the code down considerably. In this particular example convergence is very easy and neither of these limits will ever be reached, but we set them here for reference.

Submission Script

Now the hard part is done and we just need to run the model. Save the CO_oxidation.mkm file and create a new file called "mkm_job.py". This will be the submission script.

```
from catmap import ReactionModel

mkm_file = 'CO_oxidation.mkm'
model = ReactionModel(setup_file=mkm_file)
model.run()
```

If we run this file with "python mkm_job.py" then the output should look something like:

```
>> mapper_iteration_0: status - 100 points do not have valid solution.
>> minresid_iteration_0: success - [ 3.00, 4.00] using coverages from [ 3.00, 4.00]
>> minresid_iteration_0: success - [ 3.00, 3.50] using coverages from [ 3.00, 3.50]
>> ...
>> ...
>> ...
>> minresid_iteration_0: success - [-1.00, 0.00] using coverages from [-1.00, 0.00]
>> minresid_iteration_0: success - [-1.00,-0.50] using coverages from [-1.00,-0.50]
>> mapper_iteration_1: status - 0 points do not have valid solution.
```

These lines give information on where and how the solutions are converging. They are useful for debugging the model and improving convergence, but for now the only thing that matters is the final line which tells you that "0 points do not have valid solution." In other words, the solver worked!

We can run the file again (python mkm_job.py) and see that the solution is even faster this time and that the output is slightly different:

```
>> initial_evaluation: success - initial guess at point [-1.00,-0.50]
>> initial_evaluation: success - initial guess at point [-1.00, 0.00]
>> initial_evaluation: success - initial guess at point [-1.00, 0.50]
>> ...
```

As the output suggests the solution is faster because it is using the solutions from the previous run as initial guesses. Since the model has not changed the initial guesses are right (at least within 1e-100) so the solution happens very fast.

2.2.2 Analyzing the Output

Accessing Output

If you look in the working directory you should see 5 files:

- energies.txt (input file)
- CO_oxidation.mkm (setup file)
- mkm_job.py (submission script)
- CO_oxidation.log (log file)
- CO_oxidation.pkl (data file)

The log file and the data file contain all information about the solved model. The log file is human-readable. If you open it up you will notice that it is actually a python script which contains many of the same things as are found in 'CO_oxidation.mkm', but also contains a number of new variable definitions. You will also see that it automatically reads in 'CO_oxidation.pkl' and stores the variables from this pickle file in the local namespace. Thus, the "data file" is actually just an extension of the log file which is stored in binary form (this saves a lot of time since the data is often so large). There are two interesting things you can do with this log file:

Load it in as a setup_file to a ReactionModel

Make a new file called "test.py" and enter the lines:

```python
from catmap import ReactionModel

model = ReactionModel(setup_file='CO_oxidation.log')

print model.rxn_expressions
print model.coefficient_matrix
```

Notice that the rxn_expressions are identical to those from the setup file, but that the coefficient_matrix also exists even though we did not define it in the setup file. The coefficient_matrix was created by the scaler during the process of solving the model. The variable "model" in test.py is actually equivalent to the variable "model" in mkm_job.py right after the line with "model.run()". This is a useful way to load in a model which is already solved for future analysis.

View output in interactive python mode

The file can be opened and viewed interactively by entering:

```
python -i CO_oxidation.log
```

in the command line. You will now have an interactive python prompt where you can view the various outputs and attributes of the solved model. For example we can look at the coverages or rates as a function of descriptor space:

```
>>> coverage_map
[[0.7777777777777777, 2.0], [mpf('1.553678172737489e-14'), mpf('0.99999999999788455
↪')]], [[0.33333333333333326, 1.0], [mpf('0.75379752729405923'), mpf('0.
↪246202464223187')]], [[1.2222222222222223, 3.5], [mpf('9.4245829753741903e-28'),␣
↪mpf('0.99999999982984852')]], [[0.7777777777777777, 0.0], [mpf('1.0'), mpf('4.
↪0785327108804121e-19')]], [[-0.11111111111111116, 3.5], [mpf('3.6157703851402e-41'),
↪ mpf('1.0')]], [[0.33333333333333326, 0.5], .... ]
```

(continues on next page)

(continued from previous page)

```
>>> coverage_map[0]
[[0.7777777777777777, 2.0], [mpf('1.553678172737489e-14'), mpf('0.99999999999788455
↪')]]
>>> rate_map[0]
[[0.7777777777777777, 2.0], [mpf('3.0626957315361884e-11'), mpf('1.5313478657680942e-
↪11'), mpf('3.0626957315361884e-11')]]
```

The format of the "rate_map" and "coverage_map" is a list of lists where the first entry of each list is the point in descriptor space and the second is the rate/coverage. This is not particularly useful if you don't know what each number in the output corresponds to. You can find out by checking the "output_labels" dictionary:

```
>>> output_labels['coverage']
('CO_s','O_s')
>>> output_labels['rate']
([['s', 'CO_g'], ['CO_s']], [['s', 's', 'O2_g'], ['O-O_s', 's'], ['O_s', 'O_s']], [[
↪'CO_s', 'O_s'], ['O-CO_s', 's'], ['CO2_g', 's', 's']])
```

In this case the model only outputs the rate and coverage. Information on how to get more outputs can be found in *Refining a Microkinetic Model*.

Visualizing Output

Unless you possess extraordinary skills in raw data visualization then reading the raw output probably doesn't do you much good. Of course it is possible to use the raw data and write your own plotting scripts, but some tools exist within the micro-kinetics module to get a quick look at the outputs. We will explore some of these tools here.

Rate "Volcano" and Coverage Plots

Often the most interesting result from such an analysis is the so-called "volcano" plot of the reaction rate as a function of descriptor space. We can achieve this with the VectorMap plotting class (the "Vector" here refers to the fact that the rates are output as a 1-dimensional list/vector). First we instantiate the plotter using the model of interest by adding the following lines in mkm_job.py after model.run():

```
from catmap import analyze
vm = analyze.VectorMap(model)
```

Next we need to give the plotter some information on what to plot and how to plot it:

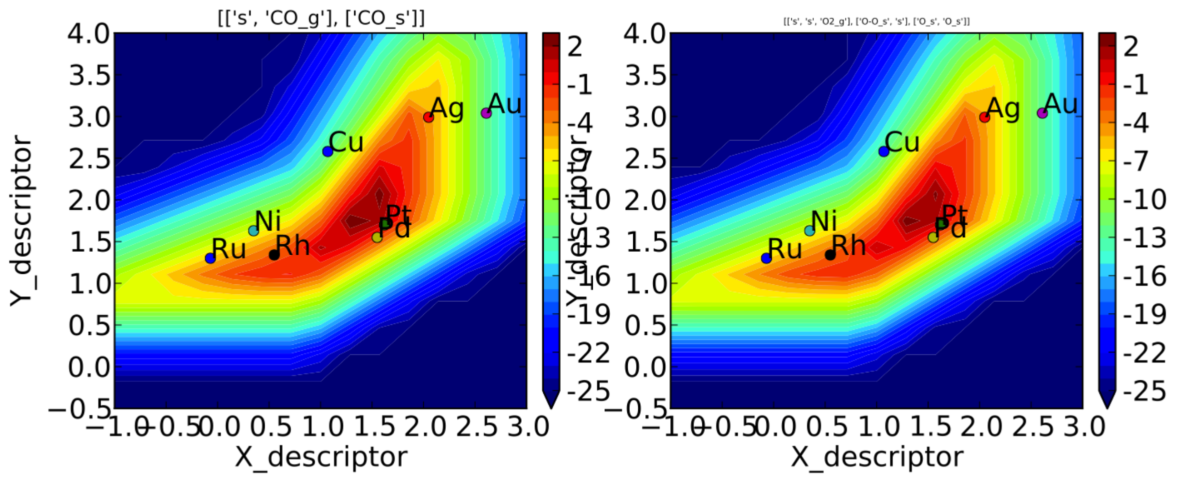
```
vm.plot_variable = 'rate' #tell the model which output to plot
vm.log_scale = True #rates should be plotted on a log-scale
vm.min = 1e-25 #minimum rate to plot
vm.max = 1e3 #maximum rate to plot
```

Most of these attributes are self-explanatory. Finally we create the plot:

```
vm.plot(save='rate.pdf') #draw the plot and save it as "rate.pdf"
```

The "`save`" keyword tells the plotter where to save the plot. You can set "`save=False`" in order to not save the plot. The plot() function returns the matplotlib figure object which can be further modified if necessary. If we run this script with "python mkm_job.py" we get the following plot:

This looks pretty similar to previously published results by Falsig et. al., with minor differences to be expected since the model and inputs used here are slightly different.



We notice that the rates are given for CO adsorption and oxygen adsorption, but that associative CO2 desorption is not included. This is because it is identical to the plot for CO adsorption (due to the steady-state condition). If we want to include it we can do:

```
vm.unique_only = False
vm.plot('all_rates.pdf')
vm.unique_only = True
```

(we turn it back to `unique_only` right afterwards since this is generally less cluttered)

which gives us a plot for each elementary step:

We might also be interested in the production rate of CO2 rather than the rates of elementary steps (it is trivial to see that they are equivalent here, but this is not always the case). If we want to analyze this we need to include the "`production_rate`" in the outputs, re-run the model, and re-plot.

```
model.output_variables += ['production_rate']
model.run()
vm.production_rate_map = model.production_rate_map #attach map
vm.threshold = 1e-30 #do not plot rates below this
vm.plot_variable = 'production_rate'
vm.plot(save='production_rate.pdf')
```

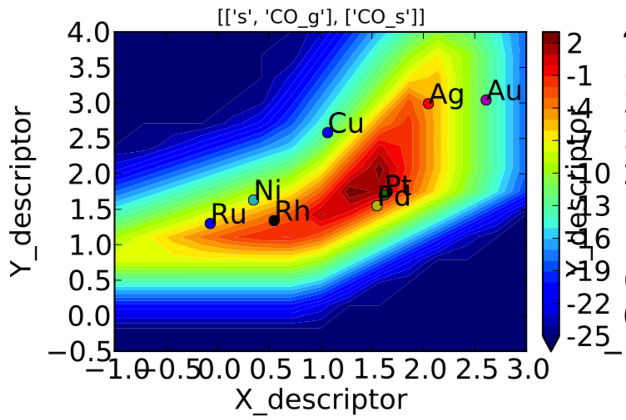
In the line commented "attach map" we point the VectorMap instance to the new output from the model. This line is not necessary if the output had been included in the original "output_variables". We also note that the "threshold" variable will be discussed in the *next tutorial*.

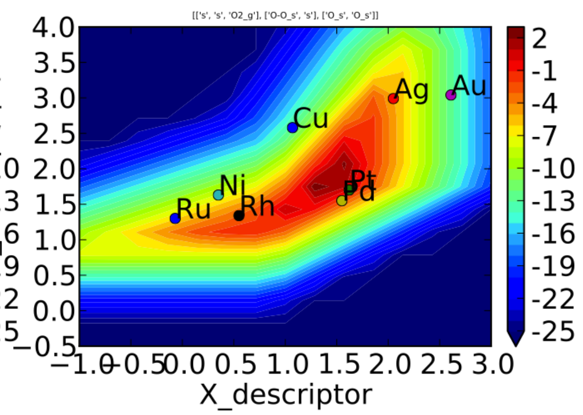
Now we can see whats going on, but its not very pretty (the colorbar is cutoff). We can make a few aesthetic improvements fairly simply:

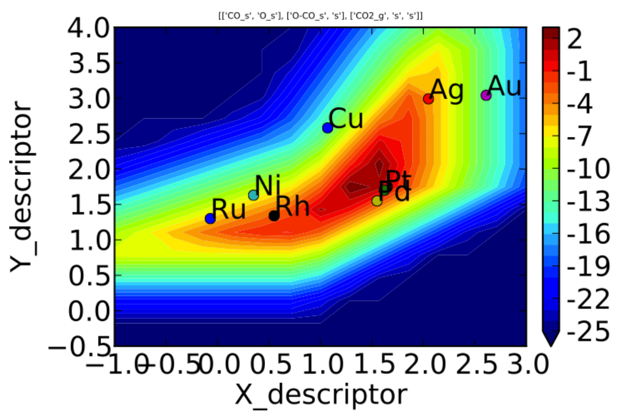
```
vm.descriptor_labels = ['CO reactivity [eV]', 'O reactivity [eV]']
vm.subplots_adjust_kwargs = {'left':0.2,'right':0.8,'bottom':0.15}
vm.plot(save='pretty_production_rate.pdf')
```

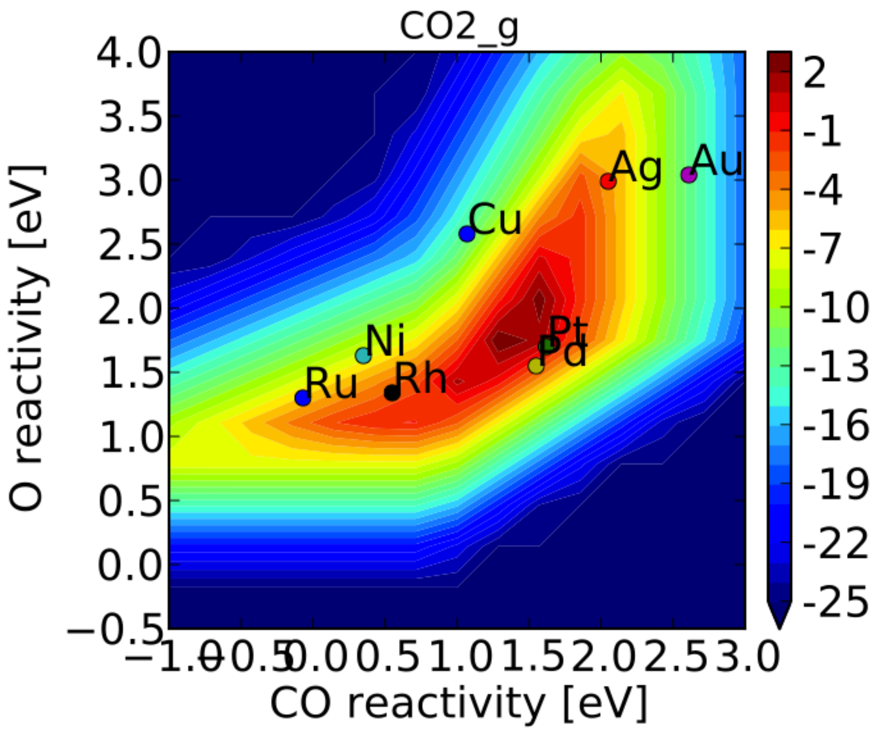
Ok, so its still not publishable, but its better. There are ways to control the finer details of the plots, but that will come in a later tutorial.

One more thing we might be interested in is the coverages of various intermediates. This can also be plotted with the VectorMap (since coverages are output as a 1-dimensional "vector"). However, we are going to want to make a few changes to the settings:

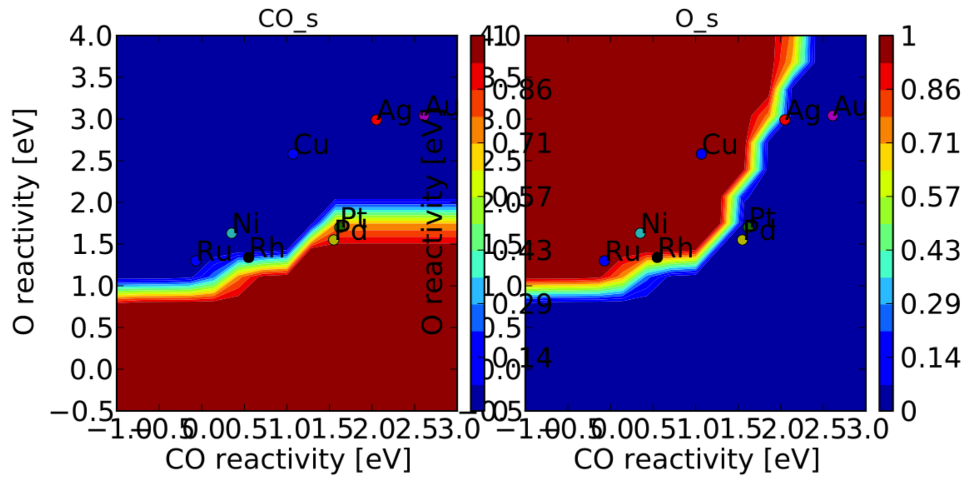

[['s', 'CO_g'], ['CO_s']]
Y_descriptor
X_descriptor
Ru
Ni
Rh
Cu
Pt
Pd
Ag
Au
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
2
-1
-4
-7
-10
-13
-16
-19
-22
-25


[['s', 's', 'O2_g'], ['O-O_s', 's'], ['O_s', 'O_s']]
X_descriptor
Ru
Ni
Rh
Cu
Pt
Pd
Ag
Au
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
2
-1
-4
-7
-10
-13
-16
-19
-22
-25


[['CO_s', 'O_s'], ['O-CO_s', 's'], ['CO2_g', 's', 's']]
Y_descriptor
X_descriptor
Ru
Ni
Rh
Cu
Pt
Pd
Ag
Au
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
2
-1
-4
-7
-10
-13
-16
-19
-22
-25



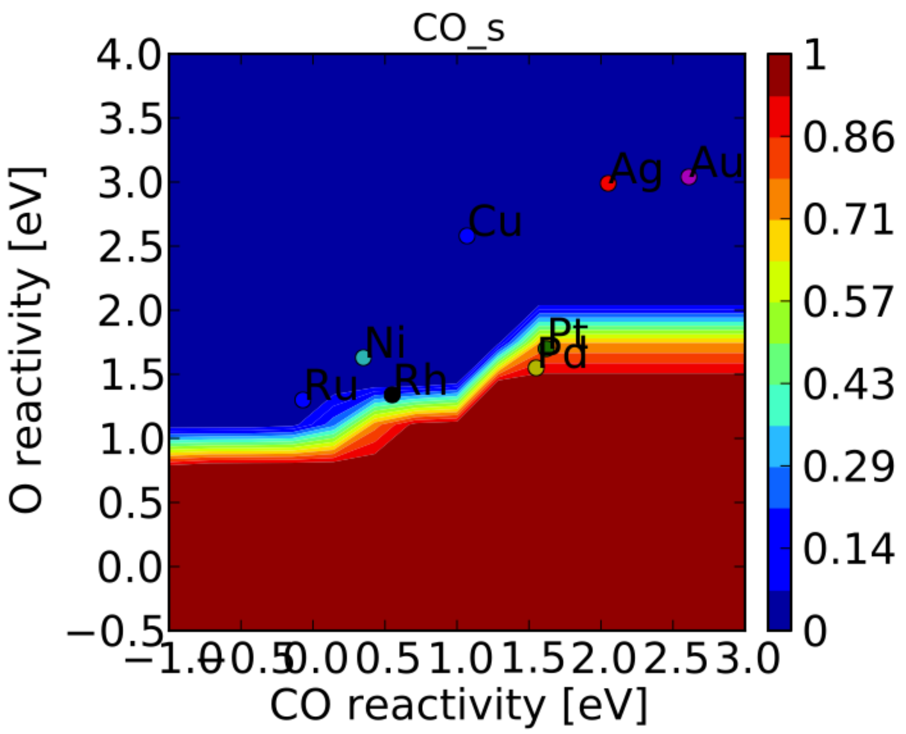
```
vm.plot_variable = 'coverage'
vm.log_scale = False
vm.min = 0
vm.max = 1
vm.plot(save='coverage.pdf')
```



Not the prettiest plot ever, but you get the point. We could re-adjust the subplots_adjust_kwargs to make this more readable, but that is left as an independent exercise.

Finally, we might not always be interested in seeing all of the coverages. If we only wanted to see the CO coverage we could specify this by:

```
vm.include_labels = ['CO_s']
vm.plot(save='CO_coverage.pdf')
```



Note that the strings to use in "`include_labels`" can be found by examining the "`output_labels`" dictionary *from the log file*; alternatively you can specify "`include_indices = [0,1,...]`" where the integers correspond to the indices of the plots to include.

2.3 Refining a Microkinetic Model

In this tutorial we will take the simple CO oxidation model presented in *Creating a Microkinetic Model* and refine it to be more complete. This tutorial should show some of the more powerful capabilities of CatMAP, and highlight the ability to dynamically make changes to the kinetic model with minimal programming. The tutorial will show several possibilities of ways to refine the model towards something that correctly represents the physical system:

- *Adding elementary steps* (refining the mechanism)
- *Adding multiple sites* (refining the active site structure)
- *Sensitivity analyses* (refining the inputs to the model)
- *Refining numerical accuracy* (resolution, tolerance, etc.)

These sections do not need to be followed sequentially. For each one we will start with the same setup file and input file from *Creating a Microkinetic Model*. We will use a slightly more efficient submission script:

```
from catmap import ReactionModel

mkm_file = 'CO_oxidation.mkm'
model = ReactionModel(setup_file=mkm_file)
```

(continues on next page)

(continued from previous page)

```
model.output_variables += ['production_rate']
model.run()

from catmap import analyze
vm = analyze.VectorMap(model)
vm.plot_variable = 'production_rate' #tell the model which output to plot
vm.log_scale = True #rates should be plotted on a log-scale
vm.min = 1e-25 #minimum rate to plot
vm.max = 1e3 #maximum rate to plot

vm.descriptor_labels = ['CO reactivity [eV]', 'O reactivity [eV]']
vm.threshold = 1e-25 #anything below this is considered to be 0
vm.subplots_adjust_kwargs = {'left':0.2,'right':0.8,'bottom':0.15}
vm.plot(save='pretty_production_rate.pdf')
```

2.3.1 Adding Elementary Steps

One way of refining a model is to include additional steps in the mechanism. To demonstrate this we will add molecular adsorption of oxygen prior to dissociation. In order to do this we need to include the relevant energetics, so add the following lines to the energies.txt:

```
Ru  111 O2  3.15    fcc []  []  "Angew. Chem. Int. Ed., 47, 4835 (2008)"
Rh  111 O2  3.63    fcc []  []  "Angew. Chem. Int. Ed., 47, 4835 (2008)"
Ni  111 O2  3.76    fcc []  []  "Angew. Chem. Int. Ed., 47, 4835 (2008)"
Pd  111 O2  4.29    fcc []  []  "Angew. Chem. Int. Ed., 47, 4835 (2008)"
Cu  111 O2  4.52    fcc []  []  "Angew. Chem. Int. Ed., 47, 4835 (2008)"
Pt  111 O2  4.56    fcc []  []  "Angew. Chem. Int. Ed., 47, 4835 (2008)"
Ag  111 O2  5.1 fcc []  []  "Angew. Chem. Int. Ed., 47, 4835 (2008)"
```

Next, we just need to define the new elementary step in the setup file (CO_oxidation.mkm):

```
rxn_expressions = [

                '*_s + CO_g -> CO*',
#               '2*_s + O2_g <-> O-O* + *_s -> 2O*',
                '*_s + O2_g -> O2_s',
                '*_s + O2_s <-> O-O* + *_s -> 2O*',
                'CO* +  O* <-> O-CO* + * -> CO2_g + 2*',

                    ]
```

Now we can run mkm_job.py to get the output. If you run in a clean directory you should see something like:

```
mapper_iteration_0: status - 225 points do not have valid solution.
minresid_iteration_0: success - [ 3.00, 4.00] using coverages from [ 3.00, 4.00]
minresid_iteration_0: success - [ 3.00, 3.68] using coverages from [ 3.00, 3.68]
minresid_iteration_0: success - [ 3.00, 3.36] using coverages from [ 3.00, 3.36]
minresid_iteration_0: success - [ 3.00, 3.04] using coverages from [ 3.00, 3.04]
minresid_iteration_0: success - [ 3.00, 2.71] using coverages from [ 3.00, 2.71]
...
minresid_iteration_1: success - [-1.00, 0.14] using coverages from [-1.00, 0.46]
rootfinding_iteration_2: fail - stagnated or diverging (residual = 3.85907297979e-29)
minresid_iteration_0: fail - [-1.00,-0.18] using coverages from [-1.00,-0.18];␣
↪initial residual was 8.73508143601e-21 (residual = 3.85907297979e-29)
minresid_iteration_1: success - [-1.00,-0.18] using coverages from [-1.00, 0.14]
```

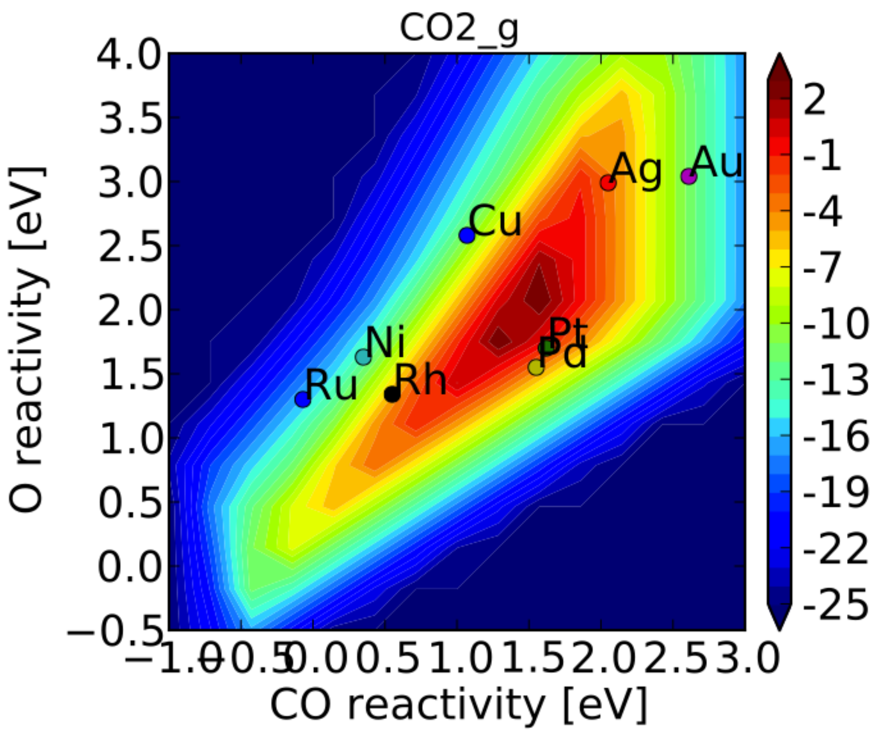
(continues on next page)

(continued from previous page)

```
minresid_iteration_0: success - [-1.00,-0.50] using coverages from [-1.00,-0.50]
mapper_iteration_1: status - 0 points do not have valid solution.
```

However, if you run in the same directory that you used for *Creating a Microkinetic Model*, you will see slightly *different output*. Either way, the model should converge.

If you look at "pretty_production_rate.pdf" it should look like the following:



If we compare this to the *figure* from the previous tutorial we can see that there are a few differences, but the general conclusions are unchanged. If we wanted to be thorough we could continue refining the model by adding more elementary steps (CO2 molecular adsorption, O-O-CO transition state, etc.). For brevity these extensions are omitted.

Using previous results as initial guesses

If you ran mkm_job.py in the same directory as you had the CO_oxidation.pkl data file from *Creating a Microkinetic Model*, you might have noticed that instead of getting output about "minresid_iterations" you get something like:

```
Length of guess coverage vectors are shorter than the number of adsorbates. Assuming␣
↪undefined coverages are 0
initial_evaluation: success - initial guess at point [ 3.00, 4.00]
Length of guess coverage vectors are shorter than the number of adsorbates. Assuming␣
↪undefined coverages are 0
initial_evaluation: success - initial guess at point [ 3.00, 3.68]
Length of guess coverage vectors are shorter than the number of adsorbates. Assuming␣
↪undefined coverages are 0
initial_evaluation: success - initial guess at point [ 3.00, 3.36]
...
```

This happens because the model detects the data file (CO_oxidation.pkl) and loads in the coverages to use as an initial guess. However, it notices that there is now more adsorbates than there are coverages since we added O2*. In order to make the best of this, it just assumes that the additional coverages are 0 and uses that as an initial guess. As you can see, it works out okay here. One thing worth noting, however, is that since the code does not know what the order of adsorbates in the previous model was, it cannot properly assign the coverages. Adsorbates are parsed in the order they appear in rxn_expressions, so in this model the order is:

```
adsorbate_names = ['CO_s','O2_s','O_s']
```

but, before adding the new elementary step the order was of course different ([‘CO_s’,’O_s’]). Since there are so few adsorbates here it turned out to be a decent initial guess that the coverage of O2* was equal to the coverage of O* from the previous model, and that the coverage of O* was 0. In general, this will not be the case. If you want to use initial guesses from previous models it is best to add the new elementary steps after the old ones. Then the new adsorbates will be assumed to have 0 coverage at the initial guess, rather than scrambling all the coverages around. This is one of the best strategies for obtaining convergence in very complex kinetic models: start with a simple version of the system and slowly add more elementary steps, converging the model along the way and using coverages from the simpler model as an initial guess to the more complex one.

More examples of how to add elementary steps are given in the *following section*.

2.3.2 Adding multiple sites

Structure dependence is a common phenomenon is catalysis, so it is important to use the correct active site structure in order to obtain accurate kinetics. Here we will look at both the (111) and (211) facets for CO oxidation using the previously defined model.

The first thing we will need to do is include the energetic inputs for (211) sites:

```
Ir  211 CO  0.673   fcc []  []  "J. Phys. Chem. C, 113 (24), 10548-10553 (2009)"
Re  211 CO  0.753   fcc []  []  "J. Phys. Chem. C, 113 (24), 10548-10553 (2009)"
Ru  211 CO  0.983   fcc []  []  "J. Phys. Chem. C, 113 (24), 10548-10553 (2009)"
Rh  211 CO  1.073   fcc []  []  "J. Phys. Chem. C, 113 (24), 10548-10553 (2009)"
Pt  211 CO  1.113   fcc []  []  "J. Phys. Chem. C, 113 (24), 10548-10553 (2009)"
Pd  211 CO  1.223   fcc []  []  "J. Phys. Chem. C, 113 (24), 10548-10553 (2009)"
Ni  211 CO  1.253   fcc []  []  "J. Phys. Chem. C, 113 (24), 10548-10553 (2009)"
Co  211 CO  1.403   fcc []  []  "J. Phys. Chem. C, 113 (24), 10548-10553 (2009)"
Fe  211 CO  1.413   fcc []  []  "J. Phys. Chem. C, 113 (24), 10548-10553 (2009)"
Cu  211 CO  2.283   fcc []   []  "J. Phys. Chem. C, 113 (24), 10548-10553 (2009)"
Au  211 CO  2.573   fcc []  []  "J. Phys. Chem. C, 113 (24), 10548-10553 (2009)"
Ag  211 CO  2.873   fcc []  []  "J. Phys. Chem. C, 113 (24), 10548-10553 (2009)"
Ru  211 O-CO    2.351   fcc []  []  "J. Phys. Chem. C, 113 (24), 10548-10553 (2009)"
Rh  211 O-CO    2.559   fcc []  []  "J. Phys. Chem. C, 113 (24), 10548-10553 (2009)"
Co  211 O-CO    2.732   fcc []  []  "J. Phys. Chem. C, 113 (24), 10548-10553 (2009)"
Ni  211 O-CO    2.768   fcc []  []  "J. Phys. Chem. C, 113 (24), 10548-10553 (2009)"
Pt  211 O-CO    3.528   fcc []  []  "J. Phys. Chem. C, 113 (24), 10548-10553 (2009)"
Cu  211 O-CO    3.918   fcc []  []  "J. Phys. Chem. C, 113 (24), 10548-10553 (2009)"
Pd  211 O-CO    3.992   fcc []  []  "J. Phys. Chem. C, 113 (24), 10548-10553 (2009)"
Ag  211 O-CO    5.099   fcc []  []  "J. Phys. Chem. C, 113 (24), 10548-10553 (2009)"
Au  211 O-CO    5.448   fcc []  []  "J. Phys. Chem. C, 113 (24), 10548-10553 (2009)"
Ag  211 O-O 5.34    fcc []  []  Falsig et al (2012)
Au  211 O-O 6.18    fcc []  []  Falsig et al (2012)
Pt  211 O-O 4.9 fcc []  []  Falsig et al (2012)
Pd  211 O-O 4.6 fcc []  []  Falsig et al (2012)
Re  211 O   -1.5    fcc []  []  Falsig et al (2012)
Co  211 O   -0.15   fcc []  []  Falsig et al (2012)
Ru  211 O   -0.1    fcc []  []  Falsig et al (2012)
```

(continues on next page)

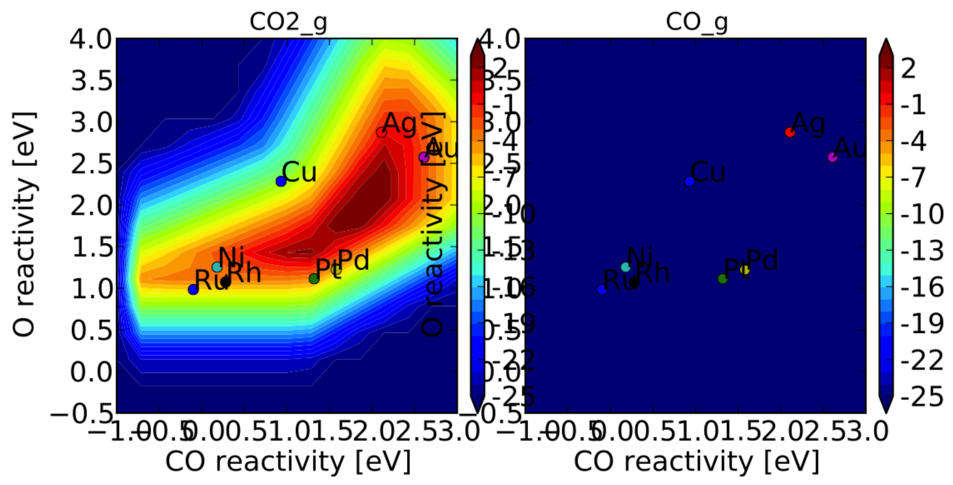
(continued from previous page)

```
Ni  211 O   0.18    fcc []  []  Falsig et al (2012)
Rh  211 O   0.28    fcc []  []  Falsig et al (2012)
Cu  211 O   0.93    fcc [] []  Falsig et al (2012)
Pt  211 O   1.32    fcc []  []  Falsig et al (2012)
Pd  211 O   1.58    fcc []  []  Falsig et al (2012)
Ag  211 O   2.11    fcc []  []  Falsig et al (2012)
Au  211 O   2.61    fcc []  []  Falsig et al (2012)
Fe  211 O   -0.73   fcc []  []  "Phys. Rev. Lett. 99, 016105 (2007)"
Ir  211 O   -0.04   fcc []  []  "Phys. Rev. Lett. 99, 016105 (2007)"
```

We note that there is no data readily available for molecular O2 adsorption on the (211) facet, so we need to make sure we move back to the simpler model from *Creating a Microkinetic Model* for the (211) analysis:

```
rxn_expressions = [

               '*_s + CO_g -> CO*',
               '2*_s + O2_g <-> O-O* + *_s -> 2O*',
#               '*_s + O2_g -> O2_s',
#               '*_s + O2_s <-> O-O* + *_s -> 2O*',
               'CO* +  O* <-> O-CO* + * -> CO2_g + 2*',

                   ]
```

If we check the "pretty_production_rate.pdf" then we see the following:

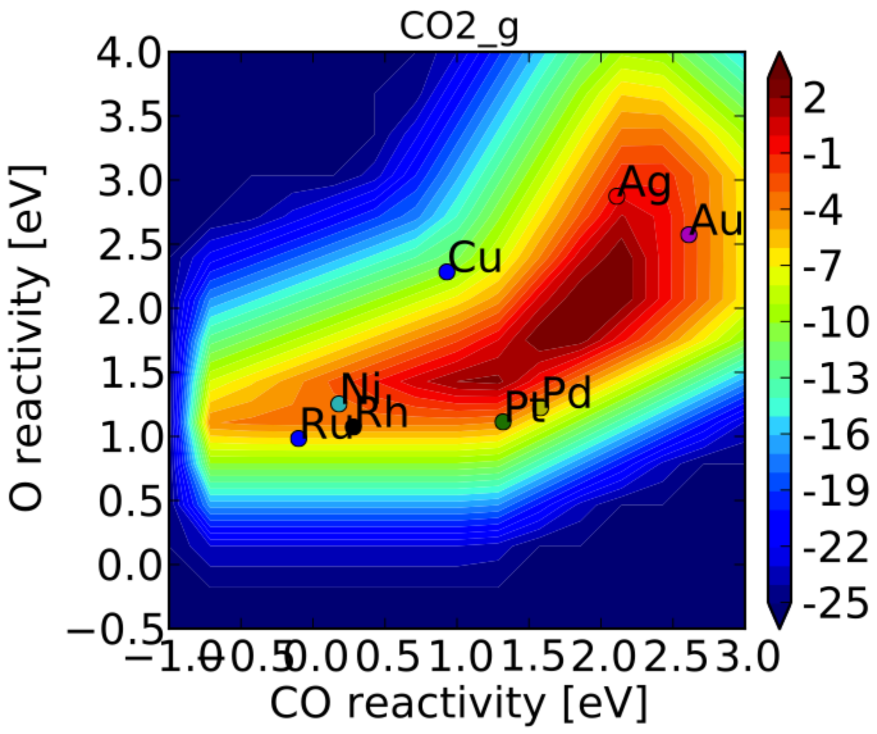


which is not very pretty. The plot on the right is showing up because the plotter says it is not empty; however, as you can see it looks pretty empty. This is happening because of numerical issues - there are some very small (<1e-50 - the tolerance) positive values for production of CO at some points in descriptor space. The quick way to get rid of this is to set a "threshold" for the plotter, so that it counts very small values as 0:

```
vm.descriptor_labels = ['CO reactivity [eV]', 'O reactivity [eV]']
vm.threshold = 1e-25
vm.subplots_adjust_kwargs = {'left':0.2,'right':0.8,'bottom':0.15}
vm.plot(save='pretty_production_rate.pdf')
```

Now we get the following:

The same thing can also be achieved by tightening the numerical precision/tolerance, as discussed *later*. When we look at the plot we see the leg going out towards Ni/Ru/Rh which, based on the *previous section*, we can predict will



be reduced if molecular oxygen adsorption is considered. We also notice that the maximum is moved towards the nobler metals, which is roughly consistent with the findings of Falsig et. al. who show that nobler metals are more active when undercoordinated clusters are examined.

Of course in a real catalyst, there will be both (111) and (211) facets (along with lots of others, but lets focus on these two for now). We can use CatMAP to examine both facets simultaneously by adding new sites. First, we need to define the mechanisms on both sites:

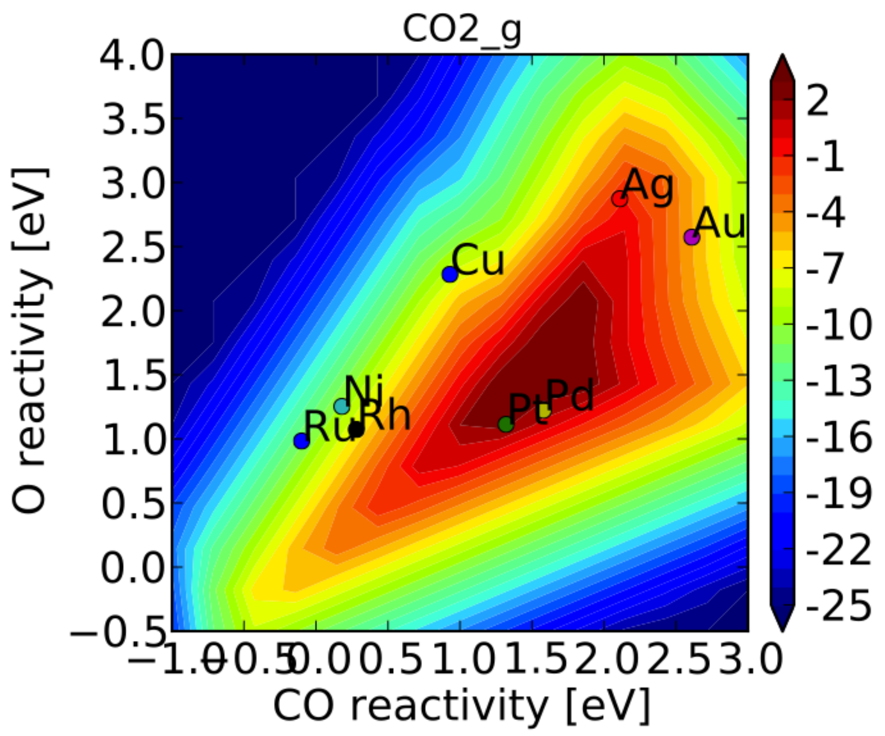
```
rxn_expressions = [

                '\*_s + CO_g -> CO*',
                '2*_s + O2_g <-> O-O* + \*_s -> 2O*',
#                '\*_s + O2_g -> O2_s',
#                '\*_s + O2_s <-> O-O* + \*_s -> 2O*',
                'CO* +  O* <-> O-CO* + * -> CO2_g + 2*',

                '\*_t + CO_g -> CO_t',
#                '2*_t + O2_g <-> O-O* + \*_t -> 2O*',
                '\*_t + O2_g -> O2_t',
                '\*_t + O2_t <-> O-O_t + \*_t -> 2O_t',
                'CO_t +  O_t <-> O-CO_t + \*_t -> CO2_g + 2*_t',

                '\*_t + CO_s -> CO_t + \*_s',
                '\*_t + O_s -> O_t + \*_s',



                    ]
```

Here we use _s (or just * which is equivalent to _s) to denote step sites, and _t to denote terrace sites. We have included molecular oxygen adsorption on the terrace, but not the step since we don't have the energetics. Diffusion between the step and terrace sites are also included, and they have no activation barrier which implies that there should be equilibrium between CO* and O* on the step/terrace. In addition to the new elementary steps, we also need to include this new "terrace site" in the species definitions:

```
species_definitions['s'] = {'site_names': ['211'], 'total':0.05} #define the sites
species_definitions['t'] = {'site_names': ['111'], 'total':0.95}
```

We also need to decide whether we want to use the (111) or (211) adsorption energies as descriptors. The proper way to do this would be to check the quality of the scaling relations and see which shows a better correlation to the parameters. However, lets just stick with the (211) sites for now.

Here we have assumed that there are 5% step sites, and 95% terrace sites. Now we can run mkm_job.py, and after a lot of fussing the model should converge. The new output looks like:



which clearly shows Pt and Pd as the best CO oxidation catalysts (as we would expect). It is a little worrying that Ag is predicted to be better than Rh, but this could be due to neglecting some mechanism (e.g. O-O-CO), neglecting zero-point and free energy contributions for adsorbates, lack of adsorbate-adsorbate interactions, or issues with the DFT input energies.

2.3.3 Free Energy Diagrams

A common way to evaluate or diagnose simpler microkinetic models is to examine the free energy diagrams that went into its creation. In the setup file, we can define any number of reaction mechanisms like the following:

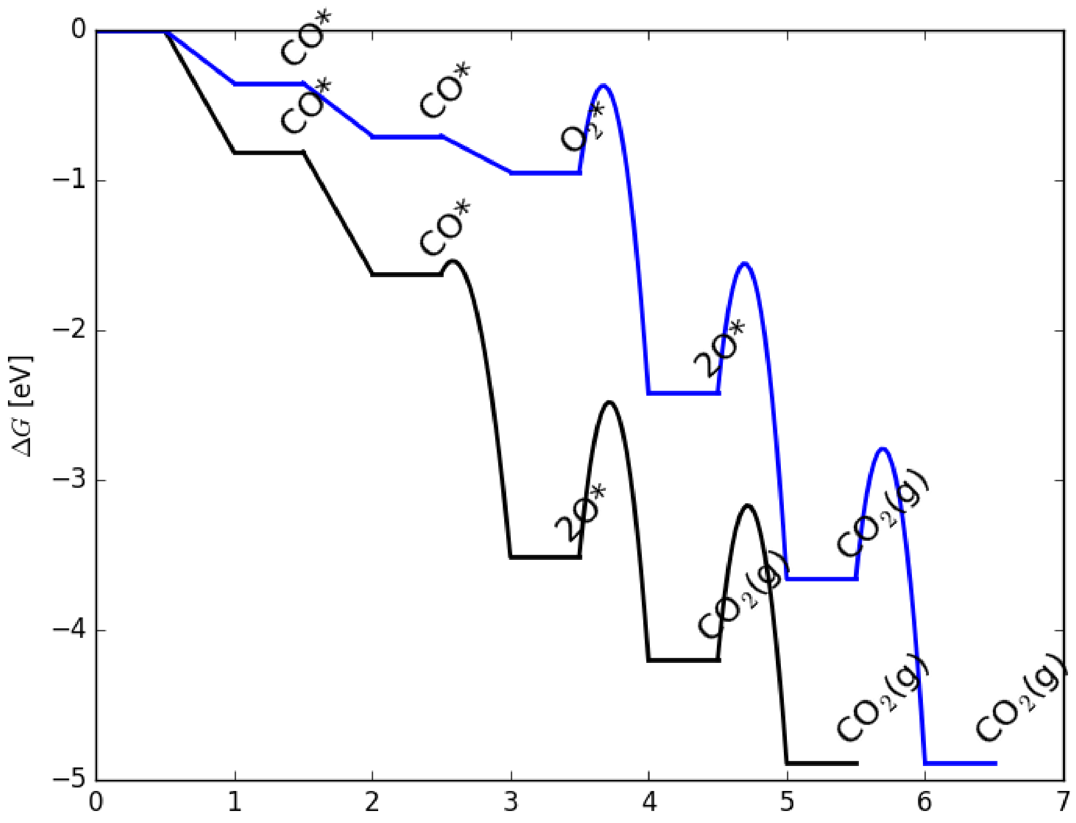
```
rxn_mechanisms = {  # these are 1-indexed
   "steps": [1, 1, 2, 3, 3],
   "terraces": [4, 4, 5, 6, 7, 7],
}
```

Here we have defined two reaction mechanisms that follow the catalytic cycle of CO oxidation on steps and terraces. The array for each mechanism is composed of the 1-indexed reaction numbers as described in `rxn_expressions`. You can use the reverse of a given elementary step by prepending the index with a negative sign.

To actually plot the free energy diagrams, we add the following lines to mkm_job.py:

```
...
ma = analyze.MechanismAnalysis(model)
ma.energy_type = 'free_energy' #can also be free_energy/potential_energy
ma.include_labels = False #way too messy with labels
ma.pressure_correction = False #assume all pressures are 1 bar (so that energies are␣
↪the same as from DFT)
ma.include_labels = True
fig = ma.plot(plot_variants=['Pt'], save='FED.png')
print(ma.data_dict)  # contains [energies, barriers] for each rxn_mechanism defined
...
```

This uses CatMAP's built-in automatic plotter to generate free energy diagrams for your defined reaction mechanisms on all surfaces by default. For clarity, we are choosing to only plot a subset of these surfaces with the `plot_variants=['Pt']` keyword argument. For electrochemical systems using ThermodynamicScaler, plot_variants instead refers to an array of voltages at which to plot free energy diagrams. The resulting plot is fairly simplistic, but feel free to generate your own nicer-looking free energy diagrams using the dictionary provided in `ma.data_dict`, which stores the values of free energies and barriers for each defined reaction mechanism.



The resulting free energy diagram is a good way to quickly determine if the results of your microkinetic model match with your expectations from its free energy inputs.

2.3.4 Sensitivity Analyses

Of course the kinetic models we are building follow the golden rule of mathematical modeling: garbage in, garbage out (i.e. your model is only as good as its inputs). Even if you have the correct mechanism and active site configuration, the results will not make sense if the data in the energy tables is inaccurate. However, in order to refine these inputs it is often useful to know which ones are most important. This can be analyzed using sensitivity analyses.

Rate Control

The degree of rate control is a powerful concept in analyzing reaction pathways. Although many varieties exist, the version published by Stegelmann and Campbell is the most general and is implemented in the micro-kinetics module. In this definition we have:

$X_{ij} = \frac{\mathrm{d}\log(r_i)}{\mathrm{d}(-G_j/kT)}$

where X_{ij} is the degree of rate control matrix, r_i is the rate of production for product *i*, G_j is the free energy of species *j*, *k* is Boltzmann's constant, and *T* is the temperature. A positive degree of rate control implies that the rate will increase by making the species more stable, while a negative degree of rate control implies the opposite.

In order to get the degree of rate control we need to add it as an output_variable in mkm_job.py:

```
...
mkm_file = 'CO_oxidation.mkm'
model = ReactionModel(setup_file=mkm_file)
model.output_variables += ['production_rate','rate_control']
model.run()
...
```

We also want to make a plot to visualize the degree of rate control:

```
mm = analyze.MatrixMap(model)
mm.plot_variable = 'rate_control'
mm.log_scale = False
mm.min = -2
mm.max = 2
mm.plot(save='rate_control.pdf')
```

The MatrixMap class is very similar to the VectorMap, except that it is designed to handle outputs which are 2-dimensional. This is true of the rate_control (and most other sensitivity analyses) since it will have a degree of rate control for each gas product/intermediate species pair. We set the min/max to -2/2 here since we know that degree of rate control is of order 1. In fact it is bounded by the number of times an intermediate appears on the same side of an elementary step. In this case that is 2, since O2* $\rightarrow$ 2O* (O* appears twice on the RHS). We could also just let the plotter decide the min/max automatically, but this is sometimes problematic due to *numerical issues with rate control*.

Now we can run the code. You should see that the initial guesses are proving successful for each point, but you will probably notice that the code is executing significantly slower (factor of ~16). The reason for this will be discussed *later*. Unlike rates/coverages, the rate control will not converge quicker with a previous solution as an initial guess. In this case it may be desirable to load in the results of a previous simulation directly like:

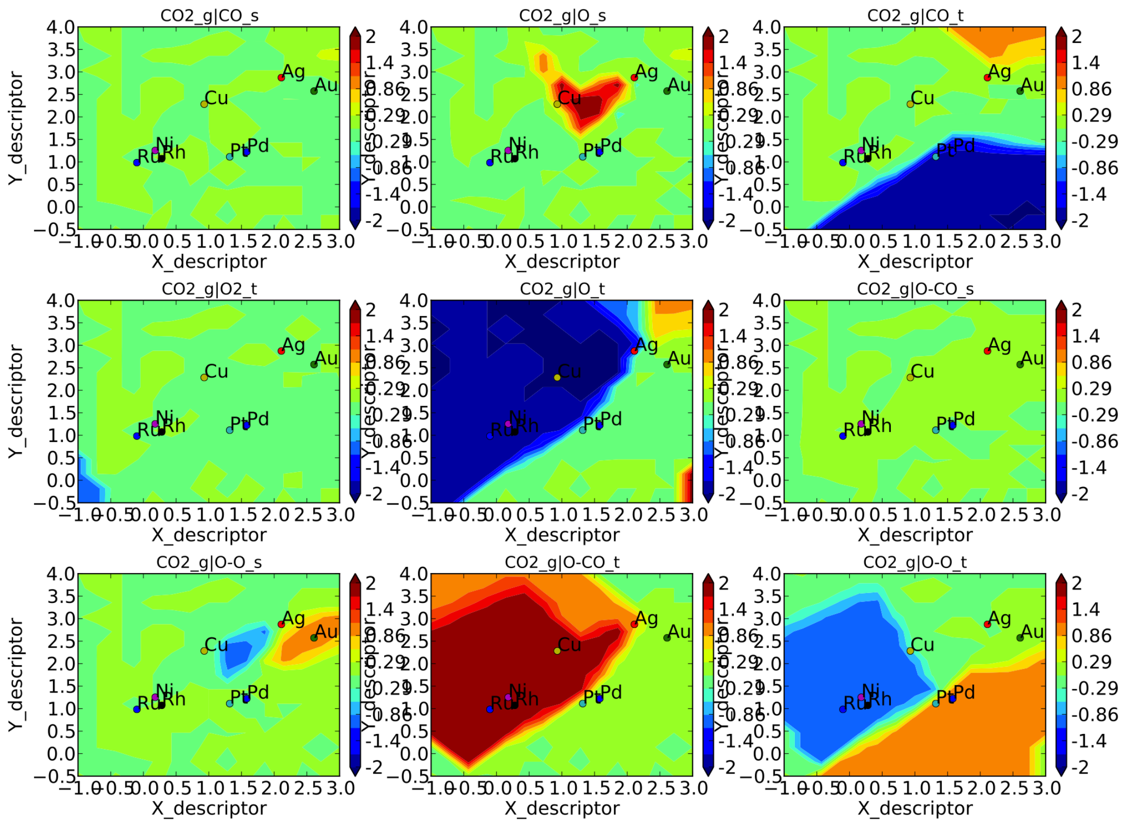
```
mkm_file = 'CO_oxidation.mkm'
#model = ReactionModel(setup_file=mkm_file)
#model.output_variables += ['production_rate','rate_control']
#model.run()
```

(continues on next page)

(continued from previous page)

```
model = ReactionModel(setup_file=mkm_file.replace('mkm','log'))
```

In general this is a good way to re-load the results of a simulation without recalculating it. Regardless, the rate control plot looks like:



This shows us that the rate is decreased when O* or CO* are bound more strongly (depending on descriptor values). Conversely, the rate can be increased by lowering the energy of the O-CO transition state, or sometimes by binding O* more strongly at the (211) site. The effect of lowering the O-O transition-state varies depending on where the surface is in descriptor space.

While these types of analyses are useful, they should be used with caution. If the energies of other intermediates change considerably then it could result in those intermediates controlling the rate. Furthermore, as discussed by Nørskov et. al, there are underlying correlations beneath the parameters, so if one wants to optimize a catalyst these must also be considered.

Other Sensitivity Analyses

Similar to the degree of rate control, the degree of selectivity control can also be defined:

$X_{ij}^{S} = \frac{\mathrm{d}\log(s_i)}{\mathrm{d}(-G_j/kT)}$

where X_{ij}^S is the degree of selectivity control, and s_i is the selectivity towards species *i*. This can be included analogously to rate control by adding 'selectivity_control' to the output variables and analyzing with the MatrixMap class.

There is also the reaction order with respect to external pressures of various gasses, given mathematically by:

$R_{ij} = \frac{\mathrm{d}\log(r_i)}{\mathrm{d}\log(pj)}$

where p_j is the pressure of gas species *j*. This can also be included in the same way as rate_control and selectivity control by including "rxn_order" in the output variables.

Numerical Issues in Sensitivity Analyses

All sensitivity analyses implemented in the micro-kinetics module are calculated via numerical differentiation. This causes them to be very slow. Furthermore, the fact that numerical differentiation is notoriously sensitive to the "infinitesimal" number used to calculate the derivative, combined with the extreme stiffness of the sets of differential equations behind the kinetic model, can lead to issues. The two most common are:

Jacobian Errors

You may sometimes notice that the model will give output like:

```
initial_evaluation: success - initial guess at point [ 2.71, 3.36]
rootfinding_iteration_3: fail - stagnated or diverging (residual = 5.22501330063e-13)
jacobian_evaluation: fail - stagnated or diverging (residual = 5.22501330063e-13).␣
↪Assuming Jacobian is 0.
initial_evaluation: success - initial guess at point [ 2.71, 3.04]
```

This implies that the coverages for the unperturbed parameters failed when used as an initial guess for the perturbed parameters. Given that the perturbation size is, by default, 1e-14, this should only happen if the system is extremely stiff. However, its not impossible. Usually you can figure out what you need to know even when you skip the points where the Jacobian fails, but in the case you really need it converged at every point, you can decrease the "perturbation_size" attribute of the reaction model. When specifying perturbations below 1e-14 it is probably a good idea to do this using the multiple-precision representation as:

```
model.perturbation_size = model._mpfloat(1e-16)
```

Diverging or Erroneous sensitivities

It is also not uncommon for the sensitivities to diverge to extremely large numbers, or just appear to be random numbers. This generally happens if the perturbation size is too small so that there is no measurable change in the values of the function. The best thing to do here is to tune the perturbation size to a slightly larger number and hope for convergence. Sometimes this does not work, in which case it might also be necessary to increase the precision and decrease the tolerance of the model by many orders of magnitude (see *Refining Numerical Accuracy*).

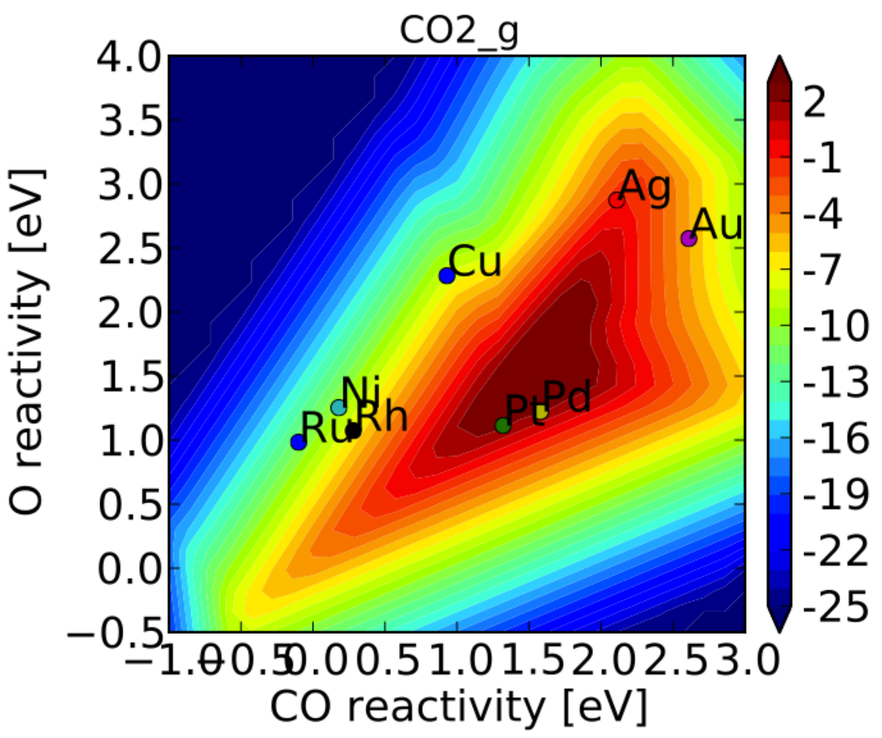
2.3.5 Refining Numerical Accuracy

A final way to refine a kinetic model is via changing the numerical parameters used for convergence, etc. A few of these parameters will be briefly discussed here:

resolution

The resolution determines the number of points between the min/max of the descriptor space. It can be a single number (same resolution in both directions) or a list the length of the number of descriptors. The latter case allows taking a higher resolution in one dimension vs. the other, which is useful if the descriptors have very different scales. It is also worth mentioning that a single-point calculation can be done by setting the resolution to 1. It is important to find a resolution that is fine enough to capture all the features in descriptor space, but of course higher resolution requires more time. It is also worth mentioning that if you want to refine the resolution it is good to pick a number like 2*old_resolution - 1 since this allows you to re-use all the points from the previous solution.

The CO oxidation volcano is shown below at a resolution of 29 (as opposed to 15):



It looks nicer, but doesn't give much new insight.

decimal_precision

This parameter represents the numerical accuracy of the kinetic model. The solutions are found using a multiple-precision representation of numbers, so it is possible to check them to "arbitrary accuracy". Of course the model will run slower as the decimal_precision is increased, but if the precision is not high enough then the results will not make sense. Generally a decimal_precision of ~100 is sufficient, but for complex models, or when the sensitivity analyses do not behave well, the decimal_precision sometimes needs to be increased upwards of 200-300 digits. If the solutions are correct it should be possible to increase this number arbitrarily and continue to quickly refine the precision of the solutions.

tolerance

The tolerance is the maximum rate which is considered 0 by the model. Thus the tolerance should be set to several orders of magnitude below the lowest rate which is relevant for the model. Usually something on the order of 1e-50 to

1e-35 is sufficient. However, when dealing with a model where the maximum rate is very low, or when trying to make sensitivity analyses more accurate, it may be necessary to decrease the tolerance to as low as 1e-decimal_precision. Similar to the decimal_precision, if the solutions are correct then it should be possible to arbitrarily decrease the tolerance (although it should never be lower than 1e-decimal_precision).

max_rootfinding_iterations

This determines the maximum number of times the Newton's method rootfinding algorithm can iterate. It is generally safe to set this to a very high number, since if the algorithm begins to diverge (or even stops converging) then it will automatically exit. Usually a number around 100-300 is practical. This parameter does not affect the solutions of the model, just if/how long the model takes to converge.

max_bisections

This determines the number of times a distance between two points in descriptor space can be "bisected" when looking for a new solution. For example, if we know the solution at (0,0) and want the solution at (0,1) the first thing to try is using the (0,0) solution as an initial guess. If that fails, the line will be bisected, and the (0,0) solutions will be tried at (0,0.5). If this fails, then (0,0.25) is tried. This continues a maximum of max_bisections times before the module gives up. This is a "desperation" parameter since it is the best way to get a model to converge, but can be very slow. It is best to start with a value of 0-3, and then slowly increase until the algorithm can find a solution at all points. If the number goes above ~6 then it is an indication that there is something fundamentally wrong with the convergence critera (i.e. the solution oscillates) and that there is no steady-state solution.

Like `max_rootfinding_iterations`, `max_bisections` will not change the overall answers to the model, but will determine if/how long it takes to converge.

CHAPTER 3

Topics

3.1 Code Overview

Descriptor based analysis is a powerful tool for understanding the trends across various catalysts. In general, the rate of a reaction over a given catalyst is a function of many parameters - reaction energies, activation barriers, thermodynamic conditions, etc. The high dimensionality of this problem makes it very difficult and expensive to solve completely and even a full solution would not give much insight into the rational design of new catalysts. The descriptor based approach seeks to determine a few "descriptors" upon which the other parameters are dependent. By doing this it is possible to reduce the dimensionality of the problem - preferably to 1 or 2 descriptors - thus greatly reducing computational efforts and simultaneously increasing the understanding of trends in catalysis.

The "catmap" Python module seeks to standardize and automate many of the mathematical routines necessary to move from "descriptor space" to reaction rates. The module is designed to be both flexible and powerful. A "reaction model" can be fully defined by a configuration file, thus no new programming is necessary to change the complexity or assumptions of a model. Furthermore, various steps in the process of moving from descriptors to reaction rates have been abstracted into separate Python classes, making it easy to change the methods used or add new functionality. The downside of this approach is that it makes the code more complex. The purpose of this guide is to explain the general structure of the code, as well as its specific functions and how to begin using it.

Useful Definitions: (Note symbols may not appear in Safari/IE)

- descriptor - variable used to describe reaction kinetics at a high level. Most commonly these are the binding energies of atomic constituents in the adsorbates (e.g. carbon/nitrogen/oxygen adsorption energies) or other intermediates. However, in the general sense other variables such as electronic structure parameters (e.g. d-band center) or thermodynamic parameters (e.g. temperature/pressure) could also be descriptors.
- descriptor space (D) - the space spanned by the descriptors. Usually $\mathbb{R}^2$.
- parameter space (P) - the space spanned by the full reaction parameters for the reaction model. Usually $\mathbb{R}^{2n}$ where n is the number of elementary steps (one reaction energy and one reaction barrier per elementary step).
- reaction rates (r) - an n-vector of rates corresponding to each of the n elementary reactions.
- descriptor map - a map of some variable (rates,coverages,etc.) as a function of descriptor space.

- reaction model - a set of elementary steps and conditions which define the physics of the kinetic system, along with the mathematical methods and assumptions used to move from "descriptor space" to reaction rates.
- setup file - a file used to define the reaction model
- input file - a file used to store other data which is likely common to many reaction models (e.g. energetics data)

Code Structure:

- The interface to the code is handled through the *ReactionModel* class. This class acts as a messenger class which broadcasts all its attributes to the other classes used in the kinetics module. The class also handles common functions such as printing reactions/adsorbates or comparing/reversing elementary steps. The attributes of *ReactionModel* are automatically synchronized with the parser, scaler, solver, and mapper so it is useful to think of *ReactionModel* as a "toolbox" where all the necessary information and common functions are stored.
- The *Parser* class serves to extend the "setup file" by reading in various quantities from an "input file". Technically the use of a parser is optional, but in practice it is extremely helpful for reading in common data such as adsorption energies or vibrational frequencies rather than re-typing them for every reaction model.

The process of creating a "descriptor map" is abstracted into three general processes, which are handled by the following classes within the kinetics module:

Scaler: Projects descriptor space into parameter space: $D \rightarrow P$

Solver: Maps parameter space into reaction rates: $P \rightarrow r$

Mapper: Moves through descriptor space. This becomes important for practical reasons since it is often necessary to use a solution at one point in descriptor space as an initial guess for a nearby point.

The *ThermoCorrections* class is responsible for applying thermodynamic corrections to the electronic energies which are used as direct inputs. This includes the contributions of entropy/enthalpy and zero-point energy due to temperature, pressure, or other thermodynamic variables.

There are also a variety of analysis classes which allow for automated analysis and visualization of the reaction model. These include the *VectorMap* and *MatrixMap* classes which create plots of outputs as a function of descriptor space. The *VectorMap* class is designed for outputs in the form of vectors (rates, coverages, etc.) while the *MatrixMap* is designed for outputs in the form of matrices (rate control, sensitivity analyses, etc.). The analysis folder also contains *MechanismAnalysis* which creates free energy diagrams, and *ScalingAnalysis* which can give a visual representation of how well the scaler projects descriptor space to parameter space.

Using the code:

Some examples can be found in the *Tutorials*, and these should explain the syntax necessary and serve as a good starting point. The currently implemented features are also briefly described below in order to allow a better understanding of the demos and creating original reaction setup files.

Using the kinetics module to conduct a descriptor analysis requires (at least) 2 files: the "setup file" which defines the reaction model, as well as another script to initialize the ReactionModel class and conduct analyses. The "setup file" is generally a static file (i.e. it is not a program) while the submission script will be an actual python program. Setup files typically end in ".mkm" for micro-kinetic model (although this is not required) while submission scripts end in .py since they are just python scripts. In addition it is very useful to also have an "input file" which contains the raw data about the energetics of the reaction model. An example of how to create an input file based on a table-like format is given in the *Generating an Input File* tutorial.

Each class described in the **Code Structure** section will require some specialized parameters. Some of these parameters are common to all variants of these classes, while others are specific to certain implementations. The possible inputs for the currently implemented variants of each class are listed below. Required attributes are +underlined+.

- *ReactionModel:*
  - rxn_expressions - These expressions determine the elementary reaction, and are the most important part of the model. They must be defined unless the elementary_rxns, adsorbate_names, transition_state_names,

and gas_names are all explicitly defined since the rxn_expressions are parsed into these 3 attributes. It is much easier to just define rxn_expressions, although it is important to note the syntax. There must be spaces between all elements of each expression (i.e. C*+O* is not okay, but C* + O* is), and species ending with _g are gasses by default. Adsorbed species may end with * or _x where * designates adsorption at the "s" site (by default), while _x designates adsorption at the "x" site (note that "x" may be any letter except "g", and that X* and X_s are equivalent). Transition-states should include a -, and reactions with a transition-state are specified by 'IS <-> TS -> FS' while reactions without a transition-state are defined as 'IS -> FS' (where IS,TS,FS are expressions for the Initial/Transition/Final State). When the model initializes it checks the expressions for mass/site balances, and if it finds that they are not balanced it will raise an exception. [list of strings]. Instead of specifying rxn_expressions the following attributes may instead be defined:

  * elementary_rxns - list version of rxn_expressions. These will be automatically populated if rxn_expressions are defined. [list of lists of lists]
  * adsorbate_names - list of adsorbate names included in the analysis. Automatically populated if rxn_expressions are defined. [list of strings]
  * transition_state_names - list of transition-state names included in the analysis. Automatically populated if rxn_expressions are defined. [list of strings]
  * gas_names - list of gas names included in the analysis. [list of strings]

- surface_names - list of surface names to be included in the analysis. [list of strings]
- species_definitions - This is a dictionary where all species-specific information is stored. The required information will vary depending on the scaler/thermo corrections/solver/mapper used, and the "parser" generally fills in most information. However, there are a few things which generally need to be supplied explicitly:

  * species_definitions[*site*]['site_names'] (where *site* is each site name in the model) - A list of "site names" which correspond to *site*. If the TableParser (default) is being used then the "site names" must also match the designations in the "site_name" column. For example, if you want the "s" site to correspond to the energetics of an adsorbate at a (211) site, and (211) sites are designated by '211' in the site_name column of the input_file, then this would be specified by: species_definitions['s'] = {'site_names':['211']}. Similarly, if you wanted the 't' site to correspond to 'fcc' or 'bridge' sites then you could specify: species_definitions['t'] = {'site_names':['fcc','bridge']}.
  * species_definitions[*site*]['total'] (where *site* is each site name in the model) - A number to which the total coverage of *site* must sum. For example, if you wanted to have a total coverage of 1 with 10% 's' sites and 90% 't' sites (with the same site definitions as above) you would specify: species_definitions['s'] = {'site_names':['211'],'total':0.1} and species_definitions['t'] = {'site_names':['fcc','bridge'],'total:0.9}.
  * species_definitions[*gas*]['pressure'] (where *gas* is each gas name in the model including the trailing _g) - The pressure of each gas species in bar. For example, if you wanted a carbon monoxide pressure of 10 bar and hydrogen pressure of 20 bar you would specify: species_definitions['CO_g']['pressure'] = 10 and species_definitions['H2_g']['pressure'] = 20. Note that for some situations you may instead need to specify a 'concentration','approach_to_equilibrium', or some other key, but in almost every situation some method for obtaining the gas pressures must be specified for each gas in the model.

- temperature - temperature used for the analysis. May not be defined if ThermodynamicScaler is being used with temperature as a descriptor. [number in Kelvin]
- descriptor_names - names of variables to be used as descriptors. [list of strings]
- descriptor_ranges - Used for mapping through descriptors space. Specify the limits of the descriptor values. Should be a list equal in length to the number of descriptors where each entry is a list of 2 floats (min and max for that descriptor). [list of lists of floats].

  - resolution - Used for mapping through descriptor space. Resolution used when discretizing over descriptor_range. [int]
  - parser - name of class to use for solver. Defaults to TableParser. [string]
  - mapper - name of class to use as a mapper. Defaults to MinResidMapper. [string]
  - scaler - name of class to use for scaler. Defaults to GeneralizedLinearScaler. [string]
  - solver - name of class to use for solver. Defaults to SteadyStateSolver. [string]
  - thermodynamics - name of class to use for thermodynamic corrections. Defaults to ThermoCorrections. [string]
  - data_file - file where large outputs will be saved as binary pickle files. Defaults to 'data.pkl' [filepath string]
  - numerical_representation - determines how to store numbers as binary. Can be 'mpmath' for multiple precision or 'numpy' for normal floats. Note that 'numpy' rarely works. Defaults to 'mpmath'. [string]

- *Parser:*
  - input_file - file where input data is stored. File must be in the correct format for the parser used. See *Generating an Input File* for more information.
- *Scaler:*
  - gas_thermo_mode - Approximation used for obtaining gas-phase free energy corrections. Defaults to ideal_gas. Other possibilities are: shomate_gas (use Shomate equation), zero_point_gas (zero-point corrections only), fixed_entropy_gas (include zero-point and assume entropy is 0.002 eV/K) , frozen_gas (no corrections), frozen_zero_point_gas (no zero-point and entropy is 0.002 eV/K). [string]
  - adsorbate_thermo_mode - Approximation used for obtaining adsorbate free energy corrections. Defaults to harmonic_adsorbate (use statistical mechanics+vibrational frequencies). Other possibilities are: hindered_adsorbate (statistical mechanics + vibrational frequencies + translational frequencies + rotational frequencies), zero_point_adsorbate (zero-point corrections only), frozen_gas (no corrections). [string]
- *Solver:*
- *SteadyStateSolver:*
  - decimal_precision - number of decimals to explicitly store. Calculation will be slightly slower with larger numbers, but will become completely unstable below some threshhold. Defaults to 50. [integer]
  - tolerance - all rates must be below this number before the system is considered to be at "steady state". Defaults to 1e-50. [number]
  - max_rootfinding_iterations - maximum number of times to iterate the rootfinding algorithm (multi-dimensional Newtons method). Defaults to 50. [integer]
  - internally_constrain_coverages - ensure that coverages are greater than 0 and sum to less than the site total within the rootfinding algorithm. Slightly slower, but more stable. Defaults to True. [boolean]
  - residual_threshold - the residual must decrease by this proportion in order for the calculation to be considered "converging". Must be less than 1. Defaults to 0.5. [number]
- *Mapper:*
- *MinResidMapper:*
  - search_directions - list of "directions" to search for existing solutions. Defaults to [[0,0],[0,1],[1,0],[0,-1],[-1,0],[-1,1],[1,1],[1,-1],[-1,-1]] which are the nearest points on the orthogonals and diagonals plus the current point. More directions increase the chances of findinga good solution, but slow the mapper down considerably. Note that the current point corresponds to an initial guess coverage provided by the solver

(i.e. Boltzmann coverages) and should always be included unless some solutions are already known. [list of lists of integers]

  - max_bisections - maximum number of time to bisect descriptor space when moving from one point to the next. Note that this is actually the number of iterations per bisection so that a total of 2max_bisections<> points could be sampled between two points in descriptor space. Defaults to 3. [integer]
  - descriptor_decimal_precision - number of decimals to include when comparing two points in descriptor space. Defaults to 2. [integer]

- *ThermoCorrections:*
  - thermodynamic_corrections - corrections to apply. Defaults to [‘gas’,’adsorbate’]. [list of strings]
  - thermodynamic_variables - variables/attributes upon which thermo corrections depend. If these variables do not change the corrections will not be updated. Defaults to [‘temperatures’,’gas_pressures’]. [list of strings]
  - frequency_dict - used for specifying vibrational frequencies of gasses/adsorbates. Usually populated by the parser. Defaults to {}. [dictionary of string:list of numbers in eV]
  - ideal_gas_params - parameters used for ase.thermochemistry.IdealGasThermo. Defaults to catmap.data.ideal_gas_params. [dictionary of string:string/int]
  - fixed_entropy_dict - entropies to use in the static entropy approximation. Defaults to catmap.data.fixed_entropy_dict. [dictionary of string:float]
  - atoms_dict - dictionary of ASE atoms objects to use for ase.thermochemistry.IdealGasThermo. Defaults to ase.structure.molecule(gas_name). [dictionary of string:ase.atoms.Atoms]
  - force_recalculation - re-calculate thermodynamic corrections even if thermodynamic_variables do not change. Slows the code down considerably, but is useful for sensitivity analyses where thermodynamic variables might be perturbed by very small amounts. Defaults to False. [boolean]
- *Analysis:*
- *MechanismAnalysis:*
  - rxn_mechanisms - dictionary of lists of integers. Each integer corresponds to an elementary step. Elementary steps are indexed in the order that they are input with 1 being the first index. Negative integers are used to designate reverse reactions. [dictionary of string:list of integers]

3.2 Accessing and reformatting output

The purpose of this tutorial is to provide an explanation of how to access the raw output of CatMAP. The plotting functions included in CatMAP are not meant to provide publication-quality figures by default, but rather to provide a quick way of analyzing the output and determining whether or not the solution is correct. If you want to create more complex plots, or more beautiful plots, then you will likely want to reformat the raw data and plot it with software of your choice (MATLAB, etc.) or at least know how to access the matplotlib figure object if you are brave enough to directly edit the figure in python.

The simplest approach to post-processing is to convert the CatMAP output into a data table and use methods you are familiar with. We will use the CO oxidation example from tutorial 3 to provide some concrete context. If you have run *Tutorial 3* you should have the file “CO_oxidation.log” in the tutorial 3 directory. Assuming you finished the tutorial, this file should have outputs for the coverage, rate, production_rate, and rate_control. We will look at how to read in each of these and convert them to more conventional tables.

First, lets take a look at how output is stored natively by CatMAP. Create a script with the following commands in the tutorial 3 directory:

```
from catmap.model import ReactionModel

model = ReactionModel(setup_file='CO_oxidation.log')
```

which reads the model (along with outputs) into the script. The model, after its run, will have “map” attributes for each output variable. The map is a python list structured like:

```
[
[[x_0,y_0],[out_0_0, out_0_1, ... , out_0_n]],
[[x_1,y_1],[out_1_0, out_1_1, ... , out_1_n]],
...,
[[x_m,y_m],[out_m_0, out_m_1, ... , out_m_n]]
]
```

where x and y represent descriptor values, and out_i_j represents the output vector at each point. For example, the coverage map on the CO oxidation model. Add the following to the script:

```
for pt, cvgs in model.coverage_map:
    print 'descriptors:', pt
    print 'coverages', cvgs
```

If you run this you will get a bunch of numbers like:

```
descriptors [1.9289202008928572, 2.232142857142857]
coverages [mpf('2.2800190488939763e-5'), mpf('0.020278026143493406'), mpf('2.
↪2031908471388691e-7'), mpf('1.4713059814195941e-5'), mpf('0.18181715627634512')]
descriptors [0.1428571428571428, 2.553571428571429]
coverages [mpf('2.7893704760253176e-27'), mpf('0.050000000000000003'), mpf('2.
↪8850535476689032e-25'), mpf('1.4466885932549904e-12'), mpf('0.94999999999855326')]
descriptors [3.0, 1.267857142857143]
coverages [mpf('0.049999987577682677'), mpf('4.4864510401468476e-25'), mpf('0.
↪69752183650972651'), mpf('4.653706806446857e-10'), mpf('1.789519988373698e-17')]
descriptors [2.1830357142857144, 2.8750000000000004]
coverages [mpf('6.4052886383458481e-12'), mpf('0.024816133631270126'), mpf('3.
↪4777261296001614e-13'), mpf('6.8031802914834851e-9'), mpf('0.32310023705530582')]
...
```

You should note that the coverages are “mpf” objects, which indicates that they are multiple precision. You probably also don’t know which intermediate each coverage corresponds to. Try the following:

```
labels = model.output_labels['coverage']
for pt,cvg in model.coverage_map:
    print 'descriptors',pt
    print 'intermediates',labels
    print 'coverages', [float(c) for c in cvg]
```

which gives the slightly more readable output:

```
descriptors [1.9289202008928572, 2.232142857142857]
intermediates ('CO_s', 'O_s', 'CO_t', 'O2_t', 'O_t')
coverages [2.2800190488939762e-05, 0.020278026143493406, 2.203190847138869e-07, 1.
↪471305981419594e-05, 0.1818171562763451]
descriptors [0.1428571428571428, 2.553571428571429]
intermediates ('CO_s', 'O_s', 'CO_t', 'O2_t', 'O_t')
coverages [2.7893704760253176e-27, 0.049999999999999996, 2.885053547668903e-25, 1.
↪4466885932549903e-12, 0.9499999999985532]
descriptors [3.0, 1.267857142857143]
```

(continues on next page)

(continued from previous page)

```
intermediates ('CO_s', 'O_s', 'CO_t', 'O2_t', 'O_t')
coverages [0.049999987577682675, 4.486451040146847e-25, 0.6975218365097264, 4.
↪653706806446857e-10, 1.7895199883736977e-17]
descriptors [2.1830357142857144, 2.8750000000000004]
intermediates ('CO_s', 'O_s', 'CO_t', 'O2_t', 'O_t')
coverages [6.405288638345848e-12, 0.024816133631270124, 3.477726129600161e-13, 6.
↪803180291483484e-09, 0.3231002370553058]
...
```

Based on this, you can probably see how to create a text table containing coverage outputs. All the other outputs follow the same basic format; however, there are a few tricky situations when looking at other outputs. For example, the “labels” for reaction-specific quantities (rates, rate constants, etc.) are actually lists which need to be flattened into strings. Even more difficult are “matrix” outputs like rate control, where the output is a list of lists rather than a single list. To make life easier I have created the following script which should create a tab-separated text table from any output (.log) file created by CatMAP. Just place this script into the output directory, and run it with the name of the output of interest as its first argument.

```python
from glob import glob
import sys
from catmap.model import ReactionModel

output_variable = sys.argv[1]
logfile = glob('*.log')
if len(logfile) > 1:
    raise InputError('Ambiguous logfile. Ensure that only one file ends with .log')
model = ReactionModel(setup_file=logfile[0])

if output_variable == 'rate_control':
    dim = 2
else:
    dim = 1

labels = model.output_labels[output_variable]

def flatten_2d(output):
    "Helper function for flattening rate_control output"
    flat = []
    for x in output:
        flat+= x
    return flat

#flatten rate_control labels
if output_variable == 'rate_control':
    flat_labels = []
    for i in labels[0]:
        for j in labels[1]:
            flat_labels.append('d'+i+'/d'+j)
    labels = flat_labels

#flatten elementary-step specific labels
if output_variable in ['rate','rate_constant','forward_rate_constant','reverse_rate_
↪constant']:
    str_labels = []
    for label in labels:
        states = ['+'.join(s) for s in label]
        if len(states) == 2:
```

(continues on next page)

(continued from previous page)

```
            new_label = '<->'.join(states)
        else:
            new_label = states[0]+'<->'+states[1]+'->'+states[2]
        str_labels.append(new_label)
    labels = str_labels

table = '\t'.join(list(['descriptor-'+d for d in model.descriptor_
↪names])+list(labels))+'\n'

for pt, output in getattr(model,output_variable+'_map'):
    if dim == 2:
        output = flatten_2d(output)
    table += '\t'.join([str(float(i)) for i in pt+output])+'\n'

f = open(output_variable+'_table.txt','w')
f.write(table)
f.close()
```

This should give you the ability to import CatMAP output into pretty much any other analysis or plotting program. However, if you are a matplotlib loyalist you may want to try to edit the figure objects directly, or perhaps even exploit the plotting capabilities of CatMAP to plot some "map" other than those created by CatMAP. For example, lets say that for whatever reason we wanted to plot the coverage of CO* times the rate of CO2 formation. We can do this by creating a python script:

```
from catmap.model import ReactionModel
from catmap.analyze import VectorMap

log_file = 'CO_oxidation.log'
model = ReactionModel(setup_file=log_file)

CO_cvg_CO2_rate_map = []
CO_idx = model.output_labels['coverage'].index('CO_s')
CO2_idx = model.output_labels['production_rate'].index('CO2_g')

for i,pt_cvg in enumerate(model.coverage_map):
    pt_rate = model.production_rate_map[i]
    pt,cvg = pt_cvg
    pt_i,rate = pt_rate
    assert pt == pt_i #ensure that points are the same

    CO_cvg = cvg[CO_idx]
    CO2_rate = rate[CO2_idx]
    CO_cvg_CO2_rate_map.append([pt,[CO_cvg*CO2_rate]]) #multiply the two and store in␣
↪new map

model.CO_cvg_CO2_rate_map = CO_cvg_CO2_rate_map #trick the model into thinking it has␣
↪this output
model.output_labels['CO_cvg_CO2_rate'] = ['theta_CO*r_CO2']

vm = VectorMap(model)
vm.plot_variable = 'CO_cvg_CO2_rate' #tell the model to plot the output you just␣
↪created
vm.log_scale = True #rates should be plotted on a log-scale
vm.min = 1e-25 #minimum rate to plot
vm.max = 1e3 #maximum rate to plot
vm.threshold = 1e-25 #anything below this is considered to be 0
```

(continues on next page)

(continued from previous page)

```
vm.subplots_adjust_kwargs = {'left':0.2,'right':0.8,'bottom':0.15}
fig = vm.plot(save='CO_cvg_CO2_rate.pdf')
```

If you run this script you will have a CatMAP-style plot of the CO* coverage multiplied by the CO2 formation rate. If you want to make post-processing modifications to the plot, then you should note that the output of the `VectorMap.plot` function is actually a `matplotlib.figure` object. You can get the handles for each axis by iterating through the `figure.axes` attribute. Sometimes it is convenient to label each axis the first time through to know which one you are editing. For example, add the following lines to the script:

```
for j,ax in enumerate(fig.axes):
    ax.annotate(str(j), [0.05,0.9], color='w', xycoords='axes fraction')

fig.savefig('figure_with_axes_labels.pdf')
```

Now if you look at the plot you will see the main axis is labeled 0, while the colorbar is 1. You can then edit the axes properties using matplotlib. Manipulating matplotlib figures is beyond the scope of this tutorial, but there is plenty of good documentation at http://matplotlib.org/.

3.3 Using Thermodynamic Descriptors

The previous tutorials have explained how to create a kinetic model which uses binding energies of key intermediates as “descriptors” in order to create a “map” of catalytic activity across different types of surfaces. This type of analysis is useful for catalyst screening, but it is also often interesting to take a closer look at a single catalyst surface and explore the effect of reaction conditions. In order to do this in CatMAP we will use the same framework as discussed in the *Code Overview*. The only difference is that instead of using energies as descriptors we will use “thermodynamic variables” (temperature, pressure for now) as descriptors. Naturally this also means that we will need to use a different “scaler” which, instead of creating a linear map uses some pre-defined expressions to add in the proper free energy contributions.

To make this clearer we will continue with the CO oxidation example and examine CO oxidation on Pt(111) as a function of temperature and pressure. Again, the purpose is not to make a publishable analysis but to become familiar with the functionality of CatMAP. We will be able to use a “submission script” (mkm_job.py) very similar to the one from *Refining a Microkinetic Model*.

```
from catmap import ReactionModel

mkm_file = 'CO_oxidation.mkm'
model = ReactionModel(setup_file=mkm_file)
model.output_variables += ['production_rate']
model.run()

from catmap import analyze

vm = analyze.VectorMap(model)
vm.plot_variable = 'production_rate' #tell the model which output to plot
vm.log_scale = True #rates should be plotted on a log-scale
vm.min = 1e-25 #minimum rate to plot
vm.max = 1e3 #maximum rate to plot
vm.threshold = 1e-25 #anything below this is considered to be 0
vm.subplots_adjust_kwargs = {'left':0.2,'right':0.8,'bottom':0.15}
vm.plot(save='production_rate.pdf')
```

For simplicity we will only look at the “production_rate” output variable in this tutorial. Other outputs can be calculated/analyzed as described in tutorials 2-3, *Creating a Microkinetic Model* and *Refining a Microkinetic Model*.

We will use a reaction model similar to the simple one used in *Creating a Microkinetic Model* since this will run faster than the more complicated models. In order to refine this to something more sophisticated you can follow the strategy outlined in *Refining a Microkinetic Model*. This also means that we can re-use the energies.txt file from *Creating a Microkinetic Model*, although we can make it even shorter since we only care about the Pt(111) energies:

```
surface_name        site_name       species_name    formation_energy        bulk_
↪structure  frequencies     other_parameters        reference
None        gas     CO2     2.45    None    [1333,2349,667,667]     []      "Angew.␣
↪Chem. Int. Ed., 47, 4835 (2008)"
None        gas     CO      2.74    None    [2170]  []      "Energy Environ. Sci., 3,␣
↪1311-1315 (2010)"
None        gas     O2      5.42    None    [1580]  []      Falsig et al (2012)
Pt  111     O       1.62    fcc     []      []      Falsig et al (2012)
Pt  111     CO      1.7     fcc     []      []      "Angew. Chem. Int. Ed., 47, 4835␣
↪(2008)"
Pt  111     O-CO    4.04    fcc     []      []      "Angew. Chem. Int. Ed., 47, 4835␣
↪(2008)"
Pt  111     O-O     5.35    fcc     []      []      Falsig et al (2012)
```

Note that we could also leave all the other energies in since the parser will just ignore any lines it doesn't need. Finally, there is the "setup file" - CO_oxidation.mkm. This is where all of the necessary changes must be made in order to use "thermodynamic descriptors". We will start with the same setup file from *Creating a Microkinetic Model*, but we need to make the following changes:

- Change the scaler Add the following line somewhere in the setup file (usually near the beginning): `scaler = 'ThermodynamicScaler'` As you can probably guess this tells CatMAP to use the "Thermodynamic-Scaler" class to move from "descriptor space" to "parameter space" (see *Code Overview*). This class will use whatever "thermo modes" are defined for gas phase/adsorbate species in order to add free energy contributions (see *Creating a Microkinetic Model*).
- Choose the relevant surface Next we need to tell CatMAP which surface to use. For this example we will look at Pt(111). To do this we just need to change the "surface_names" variable:

  ```
  surface_names = ['Pt']
  ```

  Note that the (111) surface is already selected due to the line:

  ```
  species_definitions['s'] = {'site_names': ['111'], 'total':1}
  ```

- Change the descriptors Now we need to tell the "ThermodynamicScaler" which two variables we will be using for descriptors, and we need to modify the ranges over which to vary these descriptors. Currently only temperature and pressure are implemented, although there is also an option to use log(pressure) as discussed later. For now lets look at temperatures from 400 - 1000 K and pressures from 1e-8 to 1000 bar:

  ```
  descriptor_names= ['temperature','pressure']
  descriptor_ranges = [[400,1000],[1e-8,1e3]]
  ```

- Modify temperature/pressure to be compatible - In *Creating a Microkinetic Model* we used a model where temperature and pressure were explicitly specified. This doesn't really make sense now, since we are varying these two variables. The temperature ends up not really mattering since it will be over-written as CatMAP moves through descriptor space; however, just to be unambiguous its good practice to delete the following line:

  ```
  temperature = 500 #Temperature of the reaction
  ```

Finally, we need to tell CatMAP how to handle the pressures. Previously we just defined "static pressures" for each gas-phase species, but that doesn't make sense if the total pressure is varying. In order to get around this we instead specify "concentrations" for each gas-phase species:

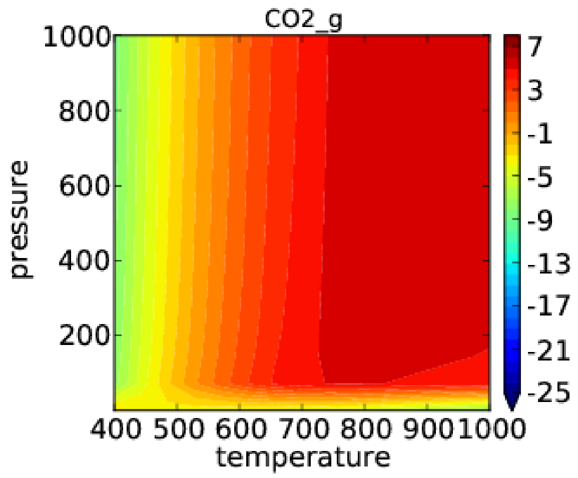
```
species_definitions['CO_g'] = {'concentration':2./3.}
species_definitions['O2_g'] = {'concentration':1./3.}
species_definitions['CO2_g'] = {'concentration':0}
```

Note that this "concentration" is not normalized - the total pressure of a gas at any total pressure will be given by concentration*P where P is the total pressure. Thus, if the concentrations do not sum to 1 then the "pressure" axis will be incorrect.

After making these changes we can run the "submission script" with:

```
sh python mkm_job.py
```

which should give the usual kind of output. When it finishes you should see the following "production_rate.pdf" in the folder:



As expected, the temperature dependence is much more drastic than the pressure dependence. In many cases it makes more sense to look at pressure dependence on a log scale. This is easily achieved by changing the descriptor names/ranges:

```
descriptor_names= ['temperature','logPressure']
descriptor_ranges = [[400,1000],[-8,3]]
```

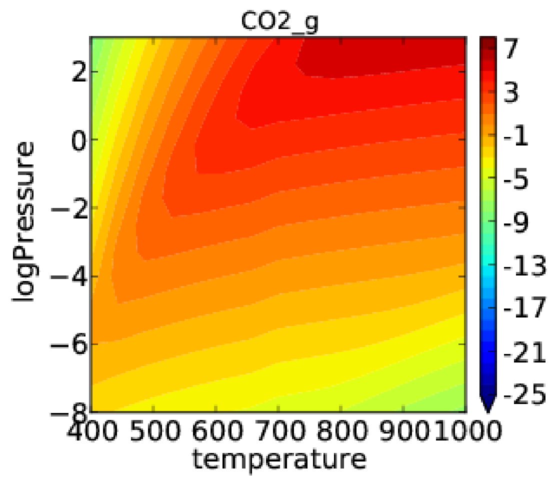
Note that the "log" in this notation refers to a base 10 logarithm so that the plot produced is the same as before, but with pressure on a log scale. If we now run the submission script we get the following output:

This looks a little nicer than the previous plot since the low pressure behavior has higher resolution.

We can see from this tutorial that it is fairly easy to move between a micro-kinetic model for a screening study and one for a "reaction condition" study (and vice-versa). Only a few lines of the "setup file" need to be changed. This is one advantage of CatMAP - once you setup a reaction model once you can re-use it for several different analyses.

3.4 Including Adsorbate-Adsorbate Interactions

Warning: The support of adsorbate-adsorbate interactions in CatMAP is currently undergoing significant changes. This will produce backwards-incompatible changes and this tutorial will change accordingly.



In all of the previous tutorials we have made some assumptions about how the adsorbed species interact with the surface lattice and each other. Specifically, we assume that every species adsorbs at a single site and does not interact with its neighbors. In this tutorial we will examine these assumptions more closely and consider some strategies for making more advanced assumptions. Please note that while most of the other features of CatMAP are relatively well tested, the features outlined in this tutorial have had very little testing/error checking so please be careful if using them in your research (and naturally let the developers list know about any issues).

It is also worth mentioning that while "ideal" mean-field models like the ones shown previously are usually mathematically well-behaved and thus exhibit "existence and uniqueness" (there is one and only one steady-state solution), all bets are off once adsorbate interactions are included. Thus it is likely possible to construct models which have oscillating solutions or no solutions at all (within the physical bounds). Naturally a system with no solution at all is unphysical, so as long as you make good assumptions then you should be okay. However, oscillating solutions are slightly trickier and it can be very hard to determine if/when this is an issue since CatMAP uses root finding to get the steady-state solution. I have yet to come across an example where CatMAP cannot find a solution to a realistic model, but be aware that it could happen.

We will continue with the CO oxidation example to avoid having to create completely new files. The tutorials will seek to demonstrate the options available within CatMAP, even if these options are not physical for the CO oxidation system, so don't take the outputs too seriously. The tutorial is divided into 2 parts which do not necessarily need to be followed sequentially:

- *Multi-site adsorbates and maximum coverages*
- *Coverage dependent adsorption energies*

For both sections we will use a submission script (mkm_job.py) similar to the one in the *Using Thermodynamic Descriptors* tutorial:

```
from catmap import ReactionModel

mkm_file = 'CO_oxidation.mkm'
model = ReactionModel(setup_file=mkm_file)
model.output_variables += ['production_rate']
model.run()

from catmap import analyze
vm = analyze.VectorMap(model)
vm.plot_variable = 'production_rate' #tell the model which output to plot
vm.log_scale = True #rates should be plotted on a log-scale
```

(continues on next page)

(continued from previous page)

```
vm.min = 1e-25 #minimum rate to plot
vm.max = 1e8 #maximum rate to plot
vm.threshold = 1e-25 #anything below this is considered to be 0
vm.subplots_adjust_kwargs = {'left':0.2,'right':0.8,'bottom':0.15}
vm.plot(save='production_rate.pdf')

vm.plot_variable = 'coverage' #tell the model which output to plot
vm.log_scale = False #coverage should not be plotted on a log-scale
vm.min = 0 #minimum coverage
vm.max = 1 #maximum coverage
vm.subplots_adjust_kwargs = {'left':0.2,'right':0.8,'bottom':0.15,'wspace':0.6}
vm.plot(save='coverage.pdf')
```

We use the same input file (energies.txt) and start with the same setup file (CO_oxidation.mkm) from *Tutorial 2* . Note that each section assumes you are starting with the "fresh" CO_oxidation.mkm file from *Tutorial 2*.

3.4.1 Multi-site adsorbates and maximum coverages

One very simple way of including adsorbate interactions is the "hard sphere exclusion" model. Actually, we have already assumed this (each site can have only 0 or 1 adsorbate), but we can also extend the assumption to allow adsorbates to occupy more than 1 site, or to set a "maximum coverage" for an adsorbate. Lets take a look at the multi-site adsorbates first:

Multi-site adsorption

CatMAP includes the ability to allow and adsorbate to adsorb to multiple sites of the same type. For instance, lets say that we want to force CO to adsorb to 2 sites. This is achieved by editing the "rxn_expressions":

```
rxn_expressions = [

                '2*_s + CO_g -> CO*',
                '2*_s + O2_g <-> O-O* + *_s -> 2O*',
                'CO* +  O* <-> O-CO* + 2* -> CO2_g + 3*',

                    ]
```

Note that we had to edit the number of sites on the CO adsorption and desorption reactions in order to make everything consistent. The next thing we need to do is tell CatMAP that CO occupies 2 sites, so that it doesn't get confused about the site balance:

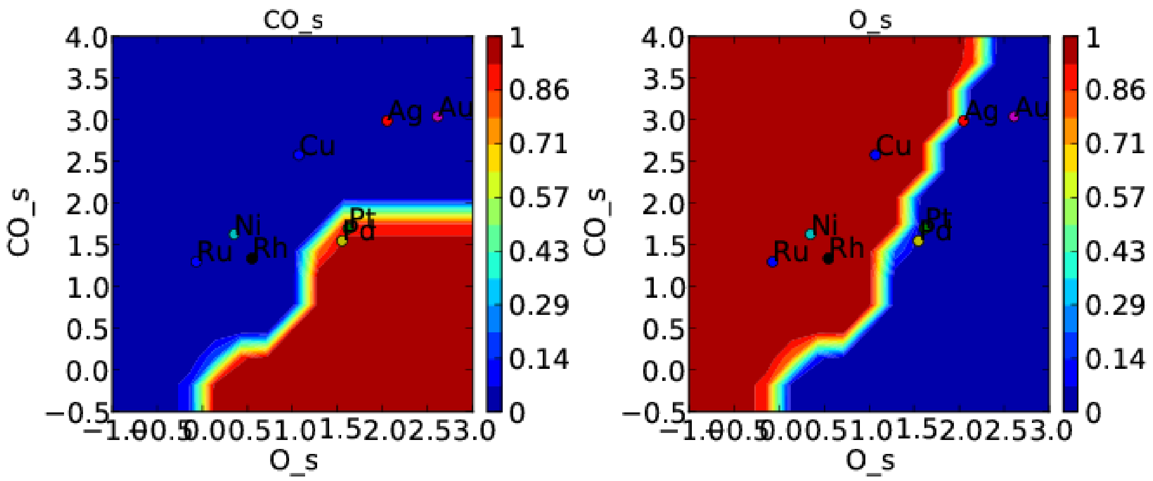
```
species_definitions['CO_s'] = {'n_sites':2}
```

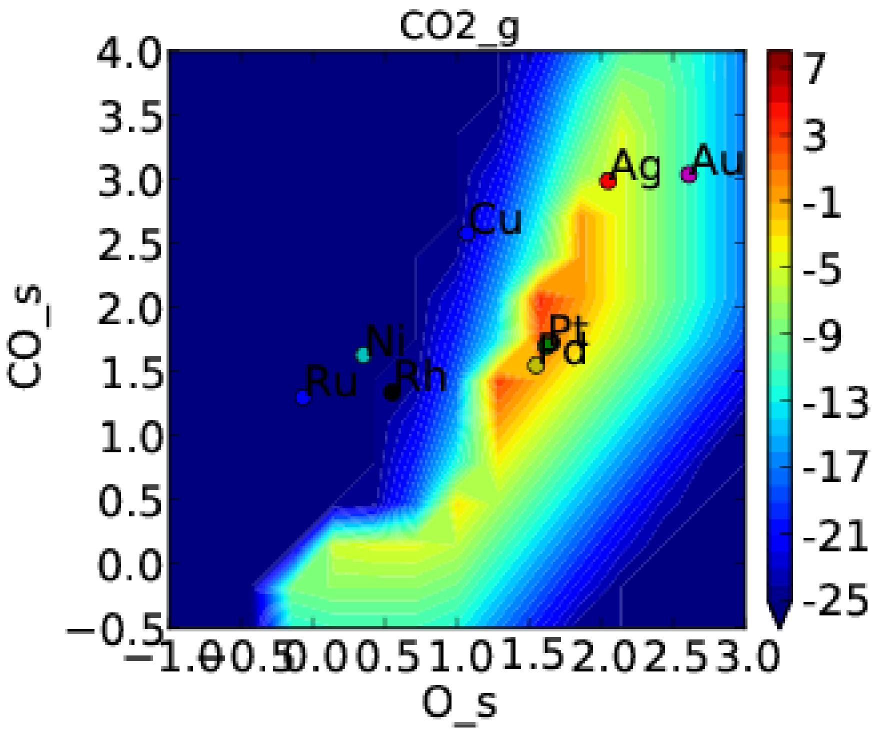
Now we can run the model and get the following coverages:

and rate:

If we compare these to *Tutorial 2* then we can see that the CO* coverage is suppressed and there is more O* in the bottom left of the plot. This is what we would expect to happen when we require an adsorbate to have an extra free site to adsorb.

There is one thing worth noting about this approach. If coverage was defined as number of CO per number of surface sites we would expect the maximum CO coverage to be 0.5 since it occupies 2 sites. However, it is clear from the plot that the coverage goes to 1. That is because we have re-defined the number of "total sites" to be a factor of 2 less for CO so that the maximum coverage of an adsorbate is always 1. This is equivalent to assuming that the probability of an adsorbate which occupies 2 sites reacting with another adsorbate is a factor of 2 higher since the site it sits on is 2


CO_s
O_s
CO_s
O_s
Ag
Au
Cu
Ni
Ru
Rh
Pt
Pd
1
0.86
0.71
0.57
0.43
0.29
0.14
0


CO2_g
CO_s
O_s
Ag
Au
Cu
Ni
Ru
Rh
Pt
Pd
7
3
-1
-5
-9
-13
-17
-21
-25

times larger. Depending on the system this may be a poor assumption, but it is the only option currently implemented in CatMAP.

Maximum coverages

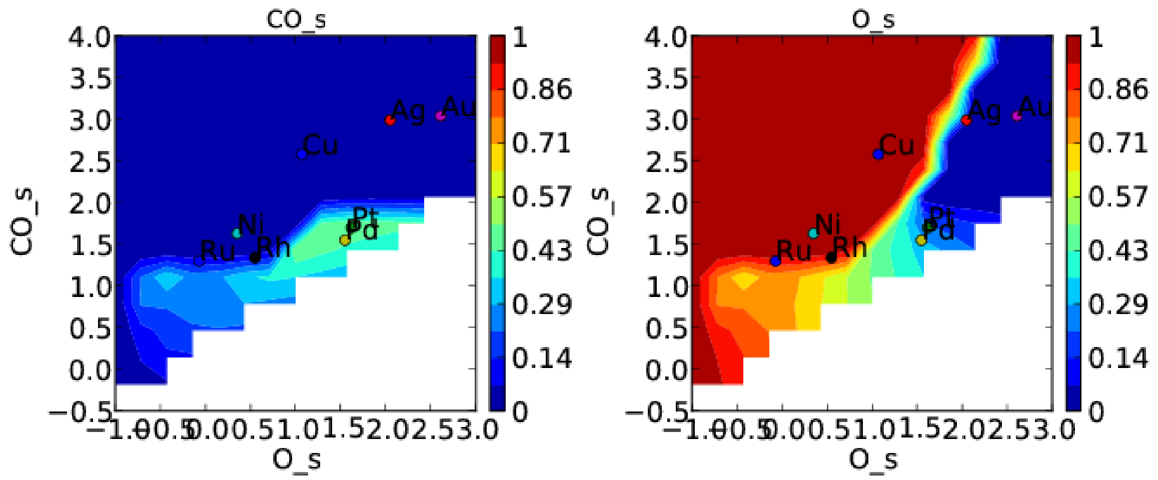
There may also be circumstances where we wish to constrain certain adsorbates to have a maximum coverage. This can easily be achieved by adding the line:

```
species_definitions['CO_s'] = {'max_coverage':0.5}
```

to CO_oxidation.mkm. However, when you run the submission script you will notice that after a lot of complaining CatMAP will give the following:

```
mapper_iteration_3: fail - no solution at 99 points.
```

This is the first time we have encountered a model that will not converge. Normally we would try to get convergence by increasing “max_bisections” or other parameters as discussed in *Tutorial 3*. However, in this case it is hopeless. This is probably because there is no solution within the bounds we have defined (which means they are not physical). This isn’t too surprising since we just made the constraint up. We can still take a look at the points that did converge in coverages.pdf:



This is pretty consistent with what we might expect. The model converges everywhere that CO coverage is less than 0.5 in the unconstrained solution, but starts to break down when the constraint limits the CO coverage to less than what is found in the unconstrained solution. Although this approach does not really make physical sense here, there could be systems where it does. In these cases CatMAP should be able to find a valid solution. Note that the “max_coverage” only pertains to one adsorbate, and does not inhibit competitive adsorption (i.e. you could have CO coverage of 0.5 and O coverage of 0.5).

3.4.2 Coverage dependent adsorption eneriges

A more powerful method for including adsorbate-adsorbate interactions is to allow adsorption energies to depend on the coverages the adsorbates. This is still relatively crude compared to an explicit lattice method like kinetic Monte Carlo, but it should provide a good picture of the first-order effects of coverage . Of course there are many ways to parameterize such a model, but there is currently only one option implemented in CatMAP - the “first order adsorption energy” model. We will first introduce the model, then look at how to use it in CatMAP, and finally show an example of how to apply it to the CO oxidation example.

First order adsorption energy model

Warning: Many instructions below this point are depricated. The first-order model itself is also not recommended, since it cannot technically be integrated. It is recommended to instead use the second-order interaction model (as described in https://pubs.acs.org/doi/abs/10.1021/jacs.5b12087) along with the "numerical_differential+integral" fitting mode, which will fit to the integral adsorption energies as well as the differences between integral adsorption energies used to compute the differential adsorption energies. However, instabilities may ensue, so it may be necessary to use only "numerical_differential" or "integral" mode instead, and the fits should be carefully manually inspected before putting too much faith in the models.

In this model we assume that adsorption energies follow the following relationship:

$$
\begin{aligned}
E_i =& E_i^i + \sum_j \mathcal{F}(|\theta|_|)\varepsilon_{\rangle|}\theta_| \\
|\theta|_j =& \sum_{\mathrm{site}_k=\mathrm{site}_j} \theta_k
\end{aligned}
$$

where E_i is the generalized formation energy for species i, |θ|j is the total coverage of occupied sites for the site on which adsorbate j is adsorbed, ε_{ij} is the "interaction matrix", and **F** is the "interaction response function" which is usually some smoothed piecewise linear function and will be discussed later. When computing the Jacobian matrix for the system we will also need the derivative of the energy with respect to coverages. This is given by:

$$
\frac{\partial E_i}{\partial \theta_l} = \sum_j \varepsilon_{ij}\left(\frac{\mathrm{d}\mathcal{F}\left(|\theta|_|\right)}{\mathrm{d}|\theta|_\mathrm{j}}\frac{\mathrm{d}|\theta|_\mathrm{j}}{\mathrm{d}\theta_\mathrm{l}}\theta_j + \mathcal{F}\left(|\theta|_|\right)\delta_{|\updownarrow}\right)
$$

The model is called "first order" since it includes only one term of coverage dependence, and this term is first order in the coverage (and $\mathcal{F}$).

We see that in order to calculate adsorption energies we need the function $\mathcal{F}$, and the matrix ε. We will also end up needing the derivative of the function $\mathcal{F}$ w.r.t. $||_j$. These two quantities will be discussed below.

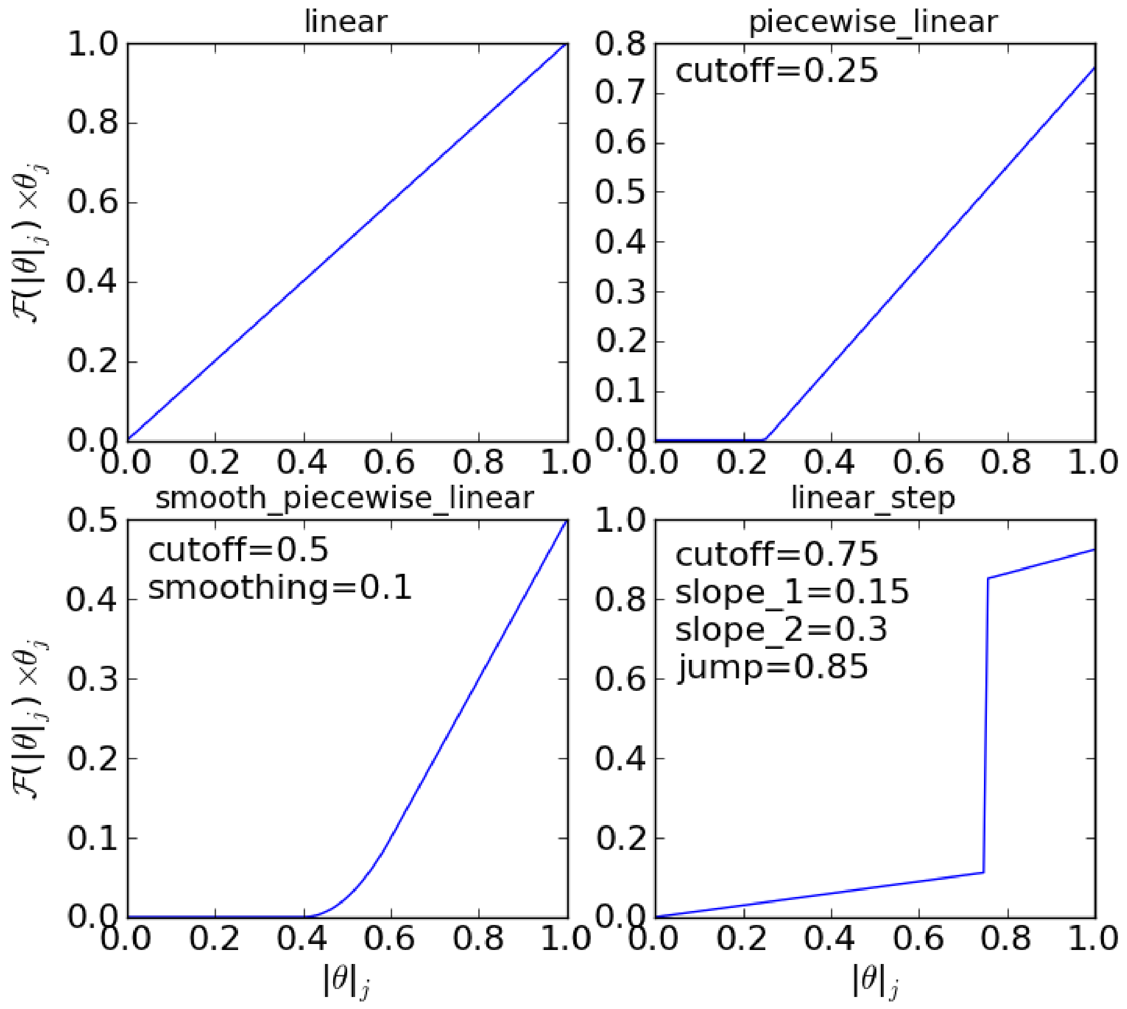
Interaction response function

The "interaction response function" determines how much the adsorption energy changes as a function of the total coverage at a site. This is necessary because adsorption energies often follow non-linear behavior as a function of coverage. Some examples of possible response functions are shown below:

The "linear", "piecewise_linear", and "smooth_piecewise_linear" are implemented in CatMAP, while the "linear_step" is a hypothetical model which could be implemented. Depending on the complexity of the interaction response function it will require some parameters. The parameters are "site specific", so that if you have a model with step sites and terrace sites you could use different "cutoffs" for the piecewise linear response function. However, the parameters do not vary by adsorbate which limits the complexity of the model.

Interaction matrix

The other key input for the "first order" interaction model is the "interaction matrix", ϵij. There are two types of terms in this matrix - "self interaction" terms (ε_{ii}) and "cross interaction" terms ($\varepsilon_{ij}(i \neq j)$). As the name suggests the "self interaction" terms tell how much an adsorbate interacts with itself, while "cross interactions" tell how much it interacts with other adsorbates. The interaction matrix is symmetric ($\varepsilon_{ij} = \varepsilon_{ji}$). The values for the matrix are determined by fitting to data. If the differential binding energies are available at various coverages then the fitting is very straightforward. However, in most cases density functional theory (DFT) will be used to calculate binding energies. Due to


linear
piecewise_linear
cutoff=0.25
smooth_piecewise_linear
cutoff=0.5
smoothing=0.1
linear_step
cutoff=0.75
slope_1=0.15
slope_2=0.3
jump=0.85
$\mathcal{F}(|\theta|_j) \times \theta_j$
$|\theta|_j$

the discreet nature of coverages in DFT, it is impossible to calculate differential binding energies. Instead, average binding energies are calculated and used to obtain the interaction parameters. The definition of average binding energy is:

$$\begin{aligned}\bar{E}_i =& \frac{\int_0^{\theta_i} E_i \mathrm{d}\theta_\mathrm{i}}{|\theta|_i} \\ =& \frac{\int_0^{\theta_i} \left(E_i^0 + \sum_k \mathcal{F}(|\theta|_\|)\varepsilon_{\rangle\|}\theta_\|\right) \mathrm{d}\theta_\mathrm{i}}{|\theta|_i}\end{aligned}$$

from this we can solve for the self-interaction parameters:

These equations look nasty at first site, but the form of $\mathcal{F}$ is usually simple enough that they aren't so intimidating. CatMAP also includes the ability to fit the self-interaction functions automatically, as discussed later.

The cross interaction terms are very costly to calculate, since they require many DFT calculations (two per adsorbate per adsorbate, or Nadsorbates2). For this reason it is common to use some approximations. The most common approximations are:

$$\begin{aligned}\text{geometric mean} : \varepsilon_{ij} =& \sqrt{|\varepsilon_{ii}\varepsilon_{jj}|} \\ \text{arithmetic mean} : \varepsilon_{ij} =& (\varepsilon_{ii} + \varepsilon_j)/2 \\ \text{neglect} : \varepsilon_{ij} =& 0\end{aligned}$$

In CatMAP the cross interaction terms are between adsorbate-adsorbate and adsorbate-transition_states. This means that the interaction matrix is actually (Nadsorbate+Ntransition-state)2. Both self and cross interactions between transition states are neglected since by definition their coverage will always be negligible. However, cross interactions between adsorbates and transition-states is not negligible. Since we don't have any self-interaction parameters for transition-states, we need some method of estimating them. This can be done by:

- transition-state scaling: transition-state scaling is used to estimate the cross parameters so that the transition-state scaling relation holds (best approximation if available)
- initial state: use the cross interaction parameters corresponding to the initial state (forward barrier is static)
- final state: use the cross interaction parameters corresponding to the final state (reverse barrier is static)
- intermediate state: use some weighted average of the initial and final state interactions (usually 0.5).
- neglect: assume to be 0 (all barriers decrease)

Implementation in CatMAP

The implementation of adsorbate-interactions requires modifications at many levels of CatMAP - specifically, the solver, scaler, and parser have been modified for the first order interaction model. However, the place that the "interactions" fit most logically into the design of CatMAP is in the "thermodynamics" since technically this is a modification of the assumption of non-interacting adsorbates. For this reason, most of the implementation has been abstracted into a class which is in the thermodynamics directory. If you are not developing then this is not a concern, but just be aware that in order to use the "first order" interaction model (or others in the future) you need to ensure that the parser, scaler, and solver are compatible. Currently the default parser (TableParser), scaler (GeneralizedLinearScaler), and solver (SteadyStateSolver) are the only ones compatible with interaction models.

Relevant Attributes

The implementation relies on the following attributes of the reaction model:

- adsorbate_interaction_model: Determines which model to use. Currently can be 'ideal' (default) or 'first_order'

- interaction_response_function: The function F from *above*. Can be 'linear', 'piecewise_linear', or 'smooth_piecewise_linear'. Can also be a callable function which takes the total coverage of a site as its first argument and the "interaction_response_parameters" dictionary as a ***kwargs* argument. Must return the value and derivative of the function at the specified total coverage.
- interaction_response_parameters: This is a dictionary of argument names/values to be used in the "interaction_response_function". The "interaction_response_parameters" can be specified as an attribute of the ReactionModel (use the same parameters for all sites) or as a key/value in the "species_definitions" dictionary for different sites (use different parameters for different sites).
- self_interaction_parameters: These are the self interaction parameters for a given adsorbate. They should be specified as a key/value in the species definition entry for the adsorbate. The key should be "self_interaction_parameters" and the value should be a list of the same length as "surface_names". The parameters should be entered for each surface in the same order that the surfaces appear in "surface_names". If an interaction parameter is not available for a surface then None should be entered.
- cross_interaction_parameters: Cross-interaction parameters can be input as a key/value pair in the species_definitions entry for one of the two adsorbates. The key should be "cross_interaction_parameters" and the value should be a dictionary where the key is the other adsorbate of the cross interaction pair and the value is a list of the same length as "surface names" where the parameters are input similar to the "self_interaction_parameters". The following is an example of how this might appear in the setup file for neglecting CO-O cross interactions on Pt, Pd, and Rh:

```
...
surface_names = ['Pt','Pd','Rh']
...
species_definitions['CO_s'] = {'cross_interaction_parameters':{'O_s':[0,0,0]}}
..
```

  Explicitly specifying cross interaction parameters is optional. Any parameters that are not explicitly specified will be estimated as specified by "cross_interaction_mode". Note that one parameter must be specified for each surface, and None can be used if a value is unknown. The "surface_names" in the example above is the same "surface_names" which defines the surfaces in the entire model, and thus should only be defined once.
- max_self_interaction: Practically it is sometimes found that the self interaction parameter should not be larger than some cutoff. This can be specified by setting the "max_self_interaction" key in the species_definitions dictionary for the adsorbate to either the numerical value or a name of one of the "surface_names" to automatically bound the interaction parameter at the value for that surface. The attribute can also be added directly to the ReactionModel in order to bound all self interaction parameters (for this, especially, using a "surface name" as a bound is recommended).
- cross_interaction_mode: The cross interaction mode tells CatMAP how to approximate cross interaction parameters that are not specified explicitly. The values can be: 'geometric_mean' (default), 'arithmetic_mean' or 'neglect' as described *above*.
- transition_state_cross_interaction_mode: Similar to "cross_interaction_mode" but for transition-states. Can be 'transition_state_scaling', 'initial_state', 'final_state', 'intermediate_state', or 'neglect' as described *above*. Using 'intermediate_state' will assume a weight of 0.5, or you can specify 'intermediate_state(X)' to set a weight of X.
- interaction_scaling_constraint_dict: The equivalent of "scaling_constraint_dict" but for interaction parameters. By default, "scaling_constraint_dict" will be used, but constraints which force slopes to be positive/negative will be removed since sign changes are expected between the "adsorbate scaling" coefficient and the interaction parameter scaling coefficient. Any parameter which does not have scaling constraints defined will be set to the "default_interaction_constraints" attribute ([None,None,None] by default).. Cross interaction parameter names are defined by 'A&B' and can appear in either order. For example, to constrain the cross parameter between O* and CO* to scale only with the first descriptor we could do:

```
interaction_scaling_constraint_dict['O_s&CO_s'] = [None,0,None]
```

Defining the constraint for 'CO_s&O_s' would be equivalent. See *Tutorial 2* for a refresher on the syntax of constraint definitions.

- non_interacting_site_pairs: Pairs of site names which are not interacting. All cross interactions between adsorbates on these sites will be set to 0. For example, to prevent cross interactions between adsorbates on the 's' and 't' site:

  ```
  non_interacting_site_pairs = [['s','t']]
  ```

  The order of adsorbates does not matter since the interaction matrix is symmetric.

- interaction_strength: All interaction parameters will be multiplied by this. Should be floatable. Defaults to 1. Useful for getting model to converge.

- interaction_fitting_mode: Determines how to construct fits to raw data. Can be None (default), 'average_self'. None implies that CatMAP should not try to automatically do any fitting because the parameters are explicitly specified. Using "average_self" will fit the self interaction parameters assuming that there are coverage-dependent average adsorption energies in the input file.

In addition, the interaction matrix can be included as an output for error-checking (this is recommended since the interaction model is still relatively new). Simply include "interaction_matrix" in the "output_variables" and analyze the output as described in *Tutorial 2 <../tutorials/creating_a_microkinetic_model>*.

CO Oxidation Example

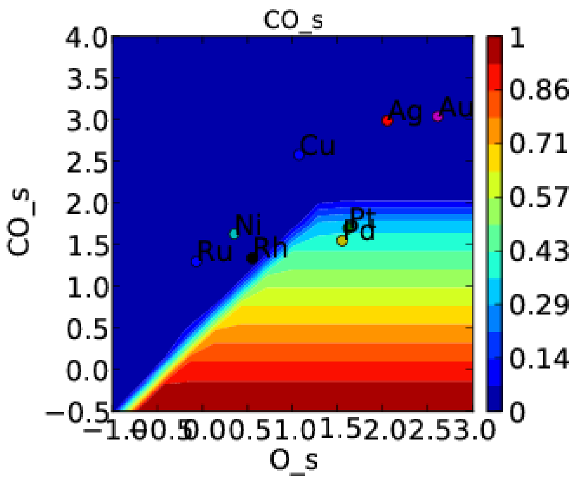
Including coverage-dependent interactions

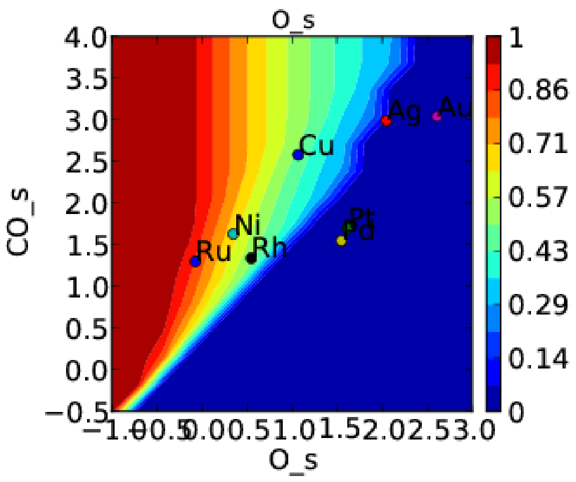
First, lets assume that we already know the self-interaction parameters and want to include coverage dependent adsorbate interactions on top of the model discussed in *Tutorial 2*. In order to do this we need to add the following to the CO_oxidation.mkm setup file:

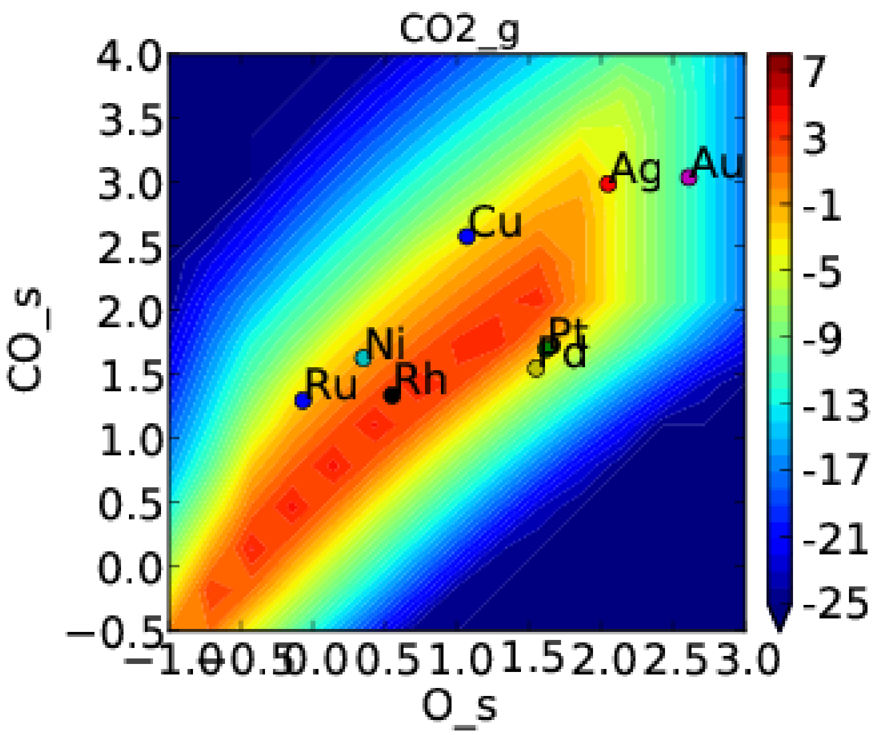
```
adsorbate_interaction_model = 'first_order' #use "first order" interaction model
interaction_response_function = 'smooth_piecewise_linear' #use "smooth piecewise␣
↪linear" interactions
species_definitions['s']['interaction_response_parameters'] = {'cutoff':0.25,
↪'smoothing':0.01}
#input the interaction paramters
#surface_names = ['Pt', 'Ag', 'Cu','Rh','Pd','Au','Ru','Ni'] #surface order reminder
species_definitions['CO_s'] = {'self_interaction_parameter':[3.248, 0.965, 3.289, 3.
↪209, 3.68, None, None, None]}
species_definitions['O_s'] = {'self_interaction_parameter':[3.405, 5.252, 6.396, 2.
↪708, 3.87, None, None, None]}
max_self_interaction = 'Pd' #self interaction parameters cannot be higher than the␣
↪parameter for Pd
transition_state_cross_interaction_mode = 'transition_state_scaling' #use TS scaling␣
↪for TS interaction
cross_interaction_mode = 'geometric_mean' #use geometric mean for cross parameters
```

If we use the same submission script as before we should get the following outputs for coverage and rate:

We can see that the coverages change much more gradually, as expected. The rate volcano is a little worrying since it now predicts Pt and Pd to be some of the worst catalysts. However, we recall that the reaction mechanism here is very simplistic, and that we are only looking at the (111) surfaces. A more realistic analysis would reveal that Pt and Pd are still the optimal catalysts, as shown by Grabow et. al..


CO_s
O_s
CO_s
Ag
Au
Cu
Ni
Ru
Rh
Pt
Pd
1
0.86
0.71
0.57
0.43
0.29
0.14
0


O_s
O_s
CO_s
Ag
Au
Cu
Ni
Ru
Rh
Pt
Pd
1
0.86
0.71
0.57
0.43
0.29
0.14
0

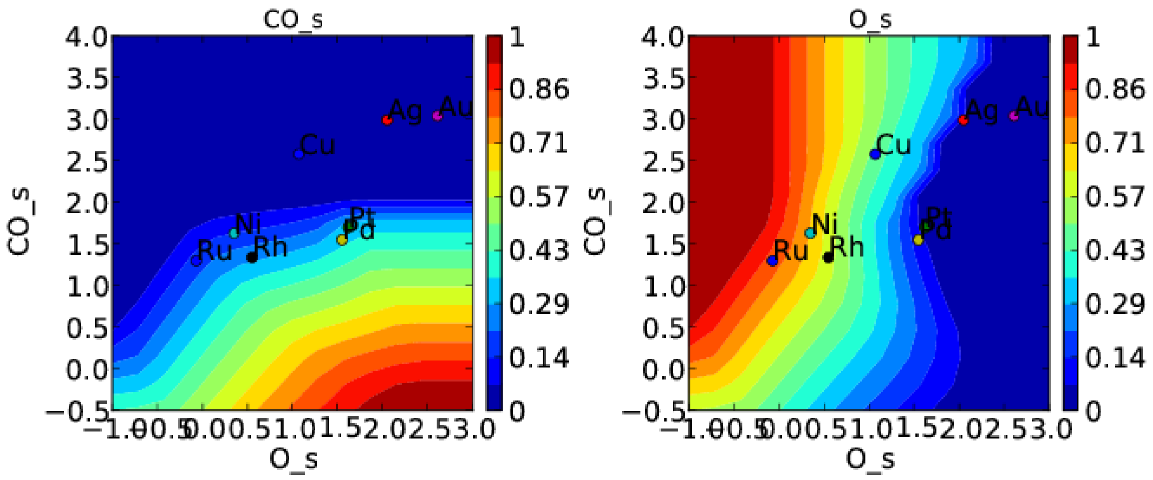

CO2_g
O_s
CO_s
Ag
Au
Cu
Ni
Ru
Rh
Pt
Pd
7
3
-1
-5
-9
-13
-17
-21
-25

Including scaled cross interactions

In the previous section we used the "geometric mean" approximation to get the cross-interaction terms from the self-interaction terms. While this is a good first approximation, it is sometimes not sufficiently accurate. In order to account for this it is possible to also include some cross-interaction terms as scaled parameters. For a very unphysical example, we will neglect cross-interactions between adsorbed O and CO, and between adsorbed CO and the O-O transition-state. This can be done by adding the following to the species definition for adsorbed CO:

```
species_definitions['CO_s'] = {'self_interaction_parameter':[3.248, 0.965, 3.289, 3.
↪209, 3.68, None, None, None],
                        'cross_interaction_parameters':{'O_s':[0,0,0,0,0,0,0,0],'O-O_s
↪':[0,0,0,0,0,0,0,0]}}
```

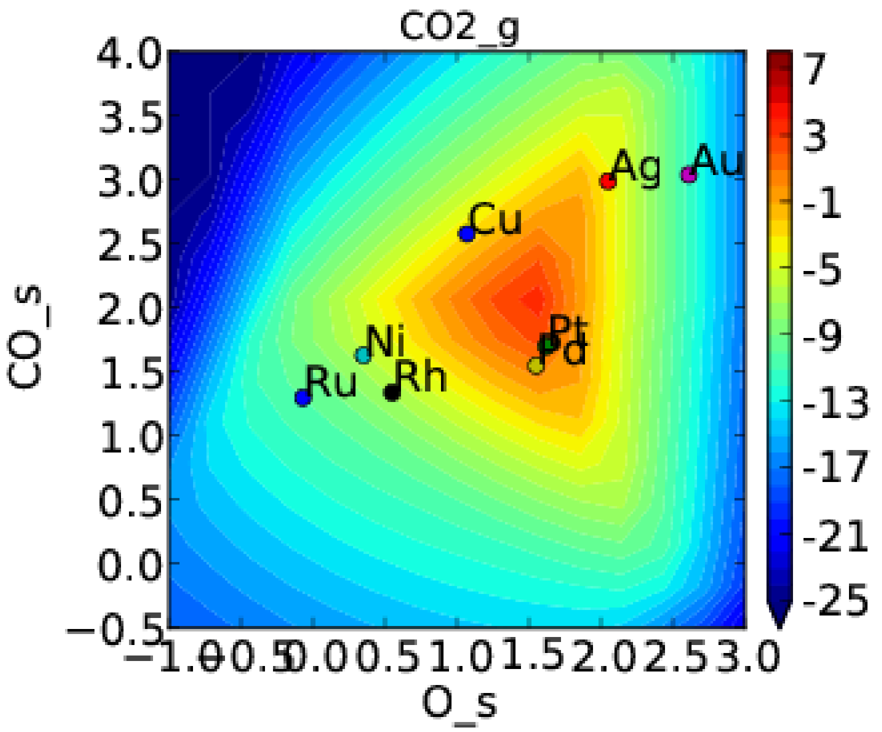
We note that the cross interactions could have equivalently been defined in the species definitions for adsorbed O and the O-O transition-states (where CO_s would be the key of the cross_interaction_parameters dictionary) but it is easier to group them both into the CO_s definition. If we now run the submission script we get the following outputs:



These results are not physical because there is no reason to expect that CO does not interact with O or O-O, but they do illustrate the syntax for specifying arbitrary cross interaction parameters. Note that the vector of zeroes here is the same length as the number of surfaces. Much like the self interaction parameters, the values of these cross interactions must be in the same order as the order of the surface names, with any unknown parameters given as None. If actual parameters were input instead of zeroes, then they would also be estimated using scaling relations in the same way the self interaction parameters are.

Using CatMAP to fit self interactions

In many cases the interaction parameters will not be available and they must be determined from some set of coverage dependent raw data. In this situation it is very convenient to have the interaction matrix automatically fit to this raw data to avoid typos and round off error in the interaction parameters. CatMAP is capable of automatically constructing this fit for the self-interaction parameters of the "first order" model described above. Fitting the second order parameters is more complicated, and should be done manually. In order to create the automatic fit it is necessary to have the energies as a function of coverage. For example, we can use the following input file with some soon to be published data for coverage dependent O and CO adsorption, along with transition-state energies from previous examples. Note that there is now a new "coverage" column:



```
surface_name    site_name   species_name    coverage    formation_energy    bulk_
↪structure  frequencies other_parameters    reference
None    gas CO2 0   2.46    None    [1333,2349,667,667] []  "NIST"
None    gas CO  0   2.77    None    [2170]  []  "Energy Environ. Sci., 3, 1311-1315␣
↪(2010)"^M
None    gas O2  0   5.42    None    [1580]  []  NIST^M
Rh  111 O   0.25    0.54    fcc []  []  Khan et. al. Parameterization of an␣
↪interaction model for adsorbate-adsorbate interactions
Pt  111 O   0.25    1.62    fcc []  []  Khan et. al. Parameterization of an␣
↪interaction model for adsorbate-adsorbate interactions
Pd  111 O   0.25    1.55    fcc []  []  Khan et. al. Parameterization of an␣
↪interaction model for adsorbate-adsorbate interactions
Cu  111 O   0.25    1.08    fcc []  []  Khan et. al. Parameterization of an␣
↪interaction model for adsorbate-adsorbate interactions
Ag  111 O   0.25    2.04    fcc []  []  Khan et. al. Parameterization of an␣
↪interaction model for adsorbate-adsorbate interactions
Au  111 O   0.25    2.75    fcc []  []  Khan et. al. Parameterization of an␣
↪interaction model for adsorbate-adsorbate interactions
Rh  111 O   0.50    0.76    fcc []  []  Khan et. al. Parameterization of an␣
↪interaction model for adsorbate-adsorbate interactions
Pt  111 O   0.50    1.9 fcc []  []  Khan et. al. Parameterization of an interaction␣
↪model for adsorbate-adsorbate interactions
Pd  111 O   0.50    1.88    fcc []  []  Khan et. al. Parameterization of an␣
↪interaction model for adsorbate-adsorbate interactions
Cu  111 O   0.50    1.755   fcc []  []  Khan et. al. Parameterization of an␣
↪interaction model for adsorbate-adsorbate interactions
Ag  111 O   0.50    2.585   fcc []  []  Khan et. al. Parameterization of an␣
↪interaction model for adsorbate-adsorbate interactions
Au  111 O   0.50    3.065   fcc []  []  Khan et. al. Parameterization of an␣
↪interaction model for adsorbate-adsorbate interactions
```

(continues on next page)

(continued from previous page)

```
Rh  111 O   0.75    1.043   fcc []  []  Khan et. al. Parameterization of an␣
↪interaction model for adsorbate-adsorbate interactions
Pt  111 O   0.75    2.243   fcc []  []  Khan et. al. Parameterization of an␣
↪interaction model for adsorbate-adsorbate interactions
Pd  111 O   0.75    2.237   fcc []  []  Khan et. al. Parameterization of an␣
↪interaction model for adsorbate-adsorbate interactions
Cu  111 O   0.75    2.423   fcc []  []  Khan et. al. Parameterization of an␣
↪interaction model for adsorbate-adsorbate interactions
Ag  111 O   0.75    3.147   fcc []  []  Khan et. al. Parameterization of an␣
↪interaction model for adsorbate-adsorbate interactions
Au  111 O   0.75    3.5 fcc []  []  Khan et. al. Parameterization of an interaction␣
↪model for adsorbate-adsorbate interactions
Rh  111 O   1.00    1.31    fcc []  []  Khan et. al. Parameterization of an␣
↪interaction model for adsorbate-adsorbate interactions
Pt  111 O   1.00    2.592   fcc []  []  Khan et. al. Parameterization of an␣
↪interaction model for adsorbate-adsorbate interactions
Pd  111 O   1.00    2.665   fcc []  []  Khan et. al. Parameterization of an␣
↪interaction model for adsorbate-adsorbate interactions
Cu  111 O   1.00    2.925   fcc []  []  Khan et. al. Parameterization of an␣
↪interaction model for adsorbate-adsorbate interactions
Ag  111 O   1.00    3.55    fcc []  []  Khan et. al. Parameterization of an␣
↪interaction model for adsorbate-adsorbate interactions
Au  111 O   1.00    3.797   fcc []  []  Khan et. al. Parameterization of an␣
↪interaction model for adsorbate-adsorbate interactions
Rh  111 CO  0.25    1.25    fcc []  []  Khan et. al. Parameterization of an␣
↪interaction model for adsorbate-adsorbate interactions
Pt  111 CO  0.25    1.49    fcc []  []  Khan et. al. Parameterization of an␣
↪interaction model for adsorbate-adsorbate interactions
Pd  111 CO  0.25    1.3 fcc []  []  Khan et. al. Parameterization of an interaction␣
↪model for adsorbate-adsorbate interactions
Cu  111 CO  0.25    2.53    fcc []  []  Khan et. al. Parameterization of an␣
↪interaction model for adsorbate-adsorbate interactions
Ag  111 CO  0.25    2.96    fcc []  []  Khan et. al. Parameterization of an␣
↪interaction model for adsorbate-adsorbate interactions
Rh  111 CO  0.50    1.58    fcc []  []  Khan et. al. Parameterization of an␣
↪interaction model for adsorbate-adsorbate interactions
Pt  111 CO  0.50    1.915   fcc []  []  Khan et. al. Parameterization of an␣
↪interaction model for adsorbate-adsorbate interactions
Ag  111 CO  0.50    3.07    fcc []  []  Khan et. al. Parameterization of an␣
↪interaction model for adsorbate-adsorbate interactions
Rh  111 CO  1.00    2.193   fcc []  []  Khan et. al. Parameterization of an␣
↪interaction model for adsorbate-adsorbate interactions
Pt  111 CO  1.00    2.473   fcc []  []  Khan et. al. Parameterization of an␣
↪interaction model for adsorbate-adsorbate interactions
Pd  111 CO  1.00    2.335   fcc []  []  Khan et. al. Parameterization of an␣
↪interaction model for adsorbate-adsorbate interactions
Cu  111 CO  1.00    3.455   fcc []  []  Khan et. al. Parameterization of an␣
↪interaction model for adsorbate-adsorbate interactions
Ag  111 CO  1.00    3.247   fcc []  []  Khan et. al. Parameterization of an␣
↪interaction model for adsorbate-adsorbate interactions
Rh  111 O-CO    0.25    3.1 fcc []  []  "Angew. Chem. Int. Ed., 47, 4835 (2008)"
Pt  111 O-CO    0.25    4.04    fcc []  []  "Angew. Chem. Int. Ed., 47, 4835 (2008)"
Pd  111 O-CO    0.25    4.2 fcc []  []  "Angew. Chem. Int. Ed., 47, 4835 (2008)"
Cu  111 O-CO    0.25    4.18    fcc []  []  "Angew. Chem. Int. Ed., 47, 4835 (2008)"
Ag  111 O-CO    0.25    5.05    fcc []  []  "Angew. Chem. Int. Ed., 47, 4835 (2008)"
Au  111 O-CO    0.25    5.74    fcc []  []  "Angew. Chem. Int. Ed., 47, 4835 (2008)"
Rh  111 O-O 0.25    3.79    fcc []  []  Falsig et al (2012)
```

(continues on next page)

(continued from previous page)

```
Pt  111 O-O 0.25    5.35    fcc []  []  Falsig et al (2012)
Pd  111 O-O 0.25    5.34    fcc []  []  Falsig et al (2012)
Cu  111 O-O 0.25    4.74    fcc []  []  Falsig et al (2012)
Ag  111 O-O 0.25    5.98    fcc []  []  Falsig et al (2012)
Au  111 O-O 0.25    7.22    fcc []  []  Falsig et al (2012)
```

Naturally the transition-states only need to be computed at a single coverage, since they do not have self interaction parameters. It is also worth noting that even if not all metals have coverage dependent data, they can still be included in the analysis (their interaction parameters will be estimated from scaling).

You can find the above data table as coverage_energies.txt in the folder for this tutorial. If you make the following changes to CO_oxidation.mkm then the parameters will be determined automatically:

```
input_file = 'coverage_energies.txt'
interaction_fitting_mode = 'average_self'
```

The "average_self" fitting mode refers to the fact that the energies in the input file are average adsorption energies, and that only the self interaction parameters will be fit. The only other option is "differential_self" which assumes that the inputs are differential adsorption energies and fits self interaction parameters.

Now, if you run mkm_job.py then you will get the same output as when the self interaction parameters were input manually (because the parameters were pre-determined by this procedure). If you want to view the parameters then you can do so by looking at the "self_interaction_parameter_dict" in the CO_oxidation.log. You should notice that they match the parameters that were input manually earlier. The advantage of the automatic fitting procedure is that any changes in the "interaction response function" will automatically be compensated for in the fit (i.e. if the smoothing value is decreased, cutoff is changed, etc.). It also makes it easier to generalize the model to inputs coming from different calculation methods, functionals, etc.

3.5 Electrochemistry

Electrochemistry in CatMAP is implemented as a set of thermodynamic corrections in the enthalpy_entropy module. To run models containing electrochemical reactions in CatMAP, the user needs to provide the following in addition to all of the typical components of a CatMAP model described in the tutorials:

- Some sort of voltage-dependent species. Currently, `pe` and `ele` are supported as fictitious gas molecules that represents the free energy of a proton-electron pair at 0V vs RHE or an electron at 0V vs SHE, respectively. As part of the Computational Hydrogen Electrode, the `pe` gas species should have the same free energy of 1/2 of a molecule of H2 gas. This species's energy (usually 0 if an H2 reference is used) can be defined manually in the input file, but defaults to half the free energy of the H2 gas molecule in your system (if using `pe`) or 0 if using "ele". These are special electrochemistry-specific gas-phase species that should ignore your choice of free energy correction scheme.
- In the reaction definitions, any potential-dependent reaction should have the `pe_g` gas molecule or `ele_g` in the initial (if reductive) or final (if oxidative) state. In addition, in the reaction definitions and input file, transition states for these reactions should contain `pe` or `ele` in the species name. For example, the reduction of adsorbed oxygen to adsorbed OH can be written as `'O* + pe_g <-> O-pe* -> OH*'`.
- An alternative formulation for electrochemical transition states is available via a special notation shown in the ORR tutorials. Instead of explicitly defining a transition state species in your input file and using it in a reaction expression, you may instead write something like `^0.26eV_a` where the 0.26eV represents the free energy barrier of elementary step at that step's equilibrium/limiting potential on site `a`.
- A voltage (vs RHE) defined as a parameter in the .mkm file e.g. `voltage = -0.2`. This is also a global thermodynamic variable, and can be manipulated as a descriptor as shown in the ORR_thermo tutorial.

- A transfer coefficient defined as a parameter in the .mkm file e.g. `beta = 0.5`. This value can be defined for each elementary step through a reaction-specific flag like `;prefactor=None, beta=0.45` at the end of the corresponding reaction expression string. See the ORR tutorials for an example of this notation.
- In the .mkm file, `electrochemical_thermo_mode` can be specified to one of three possible values (defaults to `simple_electrochemical`): `simple_electrochemical`, `hbond_electrochemical`, and `hbond_with_estimates_electrochemical`. "simple" only adds free energy corrections to adsorbates and transition states to account for voltage and beta. "hbond" will take a default or user-provided `hbond_dict` to correct each species for hydrogen bonding stabilization (see `catmap/data/hbond_data.py` for the default hbond_dict). "hbond_with_estimates" attempts to estimate a hydrogen bonding correction for a given species based on its chemical formula. This is a very crude process that is described in `catmap/thermodynamics/enthalpy_entropy.py`.

There are a few provided examples of using electrochemistry in CatMAP in the tutorials/electrochemistry directory. The HER example shows the hydrogen evolution reaction in the low-coverage limit, and the provided README file explains the details of the simplest of electrochemical reactions as done in CatMAP. The ORR example is meant to reproduce the results discussed in Hansen et al. (2014) DOI: 10.1021/jp4100608, which used a home-spun microkinetic model. The README in that directory goes into depth on how you can replicate some of the major features of this work with the provided scripts. The electron example shows how to equivalently use CatMAP for an SHE potential versus an RHE one.

3.6 Output Variables

There are a large variety of possible output variables catmap can produce as demonstrated by the "output_variables" tutorial, which provides a comprehensive list of them. You can verify that they all work by running the tutorial and inspecting the plots generated or inspect the raw data in the .pkl file. A brief description of some of the more common ones are presented here below.

- `free_energy`: contains the free energy of all species in the reaction model as calculated by the scaler. This, in conjunction with the MechanismAnalysis module can be useful in diagnosing issues in the model where the free energies of intermediates are unexpected in some way. Along with the related scaler-originated variables `rxn_parameter`, `frequency`, `electronic_energy`, `zero_point_energy`, `enthalpy`, and `entropy`, these are not typically plotted on a contour plot as they are in the tutorial without manual adjustment to the plotting parameters.
- `rate_control` and `selectivity_control`: see *Sensitivity analyses*
- `production_rate`: The rate of the production of each gas phase species. If the gas is being consumed, `consumption_rate` or `turnover_frequency` (which encapsulates both production and consumption rates) may be more useful.
- `coverage`: Coverage of each adsorbed species. Very useful for quick sanity checks of "does this model output make sense".
- `rate`: net rate (forward - backward) for each elementary step. The derived quantities `forward_rate` and `reverse_rate` are the absolute values of `rate` corresponding to the overall reaction direction. Specifically for the forward and reverse rates before they are summed, use `directional_rates`.

3.7 Additional Gas and Adsorbate Species Properties

There are two main ways of using free energies in CatMAP:

1. **Frozen everything** - Useful if you've already calculated free energies for all species yourself, you can tell CatMAP to use no free energy corrections by specifying `frozen_gas` and `frozen_adsorbate` as the `gas_thermo_mode` and `adsorbate_thermo_mode`, respectively. If you are using scaling relationships with

this method, keep in mind that linear scaling relationships are only formulated for formation energies rather than free energies.

2. **Let CatMAP do it** - CatMAP has built-in functions to estimate free energies for gasses and adsorbates based on data you provide and its own internal database of parameters. The rest of this tutorial will go over how you can customize these methods to your liking. Currently, all corrections to electronic energies to get to free energies are located in the `thermodynamics` module.

Ideal Gas Properties

`ideal_gas` is the default mode for correcting the energies of gas phase species. It relies on ASE's thermochemistry package, user-provided vibrational frequencies, and a dictionary of ideal gas parameters, `ideal_gas_params`, which stores `[symmetry_number, geometry, spin]` values for each gas species key. CatMAP provides and uses a default `ideal_gas_params` dictionary in `catmap.data.ideal_gas_params`, but you can provide your own parameters by adding something like the following to your .mkm file:

```
ideal_gas_params = {'Cl2_g':[2, 'linear', 0],
                    'F2_g':[2, 'linear', 0],}
```

and so on for each species in your system that CatMAP does not already have parameters for.

Shomate Gas Properties

In the tutorials, we use the Shomate equation to estimate free energy corrections for the gas phase molecules. Shomate parameters can be found in databases such as NIST or they can be fitted from experimental data with the `fit_shomate` function in `catmap.thermodynamics`, which takes in lists of temperatures, heat capacities, enthalpies, entropies, and initial guesses for the Shomate parameters and returns the least-squares fitted Shomate parameters A, B, C, D, E, F, G, and H. See `fit_shomate.py` in the `custom_gasses` tutorial for examples on how to do this. This tutorial also generates a plot that compares how the free energies generated from the Shomate equation and Ideal Gas equations can deviate as a function of temperature.

If you have your own Shomate parameters you wish to use or you have calculated them using `fit_shomate`, you can use them in your microkinetic model by modifying your model's `shomate_params` attribute - which is a dictionary of `gas_name:temperature_range` keys to a list of Shomate parameters as values. You can input this dictionary in your .mkm file like the example below:

```
shomate_params = {'CH3OH_g:298-1500':[-1.0846, 153.2464, -79.5305, 16.4713, 0.5220, -
↪4.8974, 199.1894, 0.0],
                  'CH3CH2OH_g:298-1200':[-4.7368, 271.9618, -169.3495, 43.7386, 0.
↪2464, -9.8283, 203.3326, 0.0],}
```

Hindered Adsorbate Properties

`hindered_adsorbate` is a mode for correcting the energies of adsorbed species. It relies on ASE's thermochemistry package HinderedThermo, user-provided vibrational frequencies, and a dictionary of hindered adsorbate parameters, `hindered_ads_params`, which stores `[barrierT, barrierR, site_density, rotational_minima, mass, inertia, symmetry_number]` values for each adsorbate species key. You can provide your own parameters by adding something like the following to your .mkm file:

```
hindered_ads_params = {'CH4_s':[0.006, 0.0008, 1.5e15, 6, None, None, 1],
                       'C2H6_s':[0.049, 0.018, 1.5e15, 6, 30.07, 73.15, 1],}
```

and so on for each adsorbed species in your system.

Contributing Data to CatMAP

If you have Shomate data or additional Ideal Gas parameter data that you'd like to see included in CatMAP, feel free to send us a pull request with your updated `parameter_data.py` file. We will most likely only include such new data in the main CatMAP repository only if the Shomate parameters are fit with well-established thermodynamic data (such as NIST or the CRC) or if the Ideal Gas parameters are consistent with the geometries in ASE's database.

Writing Your Own Corrections

Modifying CatMAP source code to include your own custom free energy corrections may seem daunting, but it may be much easier than you think. Check out the documentation for `enthalpy_entropy.py`, specifically how `ideal_gas` and `shomate_gas` are implemented. All a thermodynamic correction needs to do is to return a dictionary of corrections it is reponsible for. At the point your correction function is called, the ReactionModel instance is already fully initialized, so you have access to all attributes of the reaction model from within your correction function.

3.8 Developer Info

To get started developing you obviously need git installed. You can make changes in your local directory, and if you make a change that you think is substantial and general enough that others would be interested you can "push" the change using git (see the git tutorial). If you are going to submit a change, please update the version number in catmap/**init**.py (see **version** == x.x.x). The first number in the version is used for major changes, and a 0 represents code which is still not completely stable. The second number represents moderate changes (significant new functionality added), and the final number corresponds to the git submission number. The final number is the most annoying to update, since you need to change it every time before you submit. In order to make this slightly easier you can use the following script:

```python
from subprocess import Popen, PIPE

output = Popen(['git', 'rev-list', 'HEAD','--count'],stdout=PIPE)
rev = output.stdout.read().strip()
init = open('catmap/__init__.py')
init_txt = init.read()
init.close()
new_txt = []
for li in init_txt.split('\n'):
    if '__version__' in li:
        oldversion = li.rsplit('.',1)[0]
        newversion = oldversion + '.' + rev + '"'
        new_txt.append(newversion)
    else:
        new_txt.append(li)

new_init = '\n'.join(new_txt)
init = open('catmap/__init__.py','w')
init.write(new_init)
init.close()

message = raw_input('Submit message:')
os.system('git commit -am "'+message+'"')
os.system('git push')
```

Place the script in the base directory of the project and run it with python. If anyone knows a better way of keeping the version number synchronized I am open to suggestions.

CHAPTER 4

Reference

4.1 Subpackages

4.1.1 catmap.analyze package

Submodules

catmap.analyze.analysis_base module

catmap.analyze.matrix_map module

catmap.analyze.mechanism module

catmap.analyze.scaling module

catmap.analyze.vector_map module

Module contents

4.1.2 catmap.data package

Submodules

catmap.data.hbond_data module

`catmap.data.hbond_data.`**`generate_hbond_dict`**`()`

Returns a dictionary of generic, surface-agnostic hydrogen bond stabilizations, in eV.

catmap.data.parameter_data module

catmap.data.regular_expressions module

catmap.data.templates module

Module contents

4.1.3 catmap.mappers package

Submodules

catmap.mappers.mapper_base module

Class for 'mapping' equilibrium coverages and rates through descriptor space. This class acts as a base class to be inherited by other mapper classes, but is not functional on its own.

get_rxn_parameter_map(descriptor_ranges,resolution): Uses a scaler object to determine the reaction parameters as a function of descriptor space. May be useful for debugging or providing intuition about rate determining steps. Should return a list of the form

[[descriptor_1,descriptor_2,. . .],[rxn_parameter1, rxn_parameter2, . . .]]

save_map(map,map_file): creates a pickle of the "map" list and dumps it to the map_file

load_map(map_file): loads a "map" list by loading a pickle from the map_file

A functional derived mapper class must also contain the methods:

get_coverage_map(descriptor_ranges,resolution): a function which returns a list of the form [[descriptor_1,descriptor_2,. . .], [cvg_ads1,cvg_ads2,. . .]]

get_rates_map(descriptor_ranges,resolution): a function which returns a list of the form [[descriptor_1,descriptor_2,. . .], [rate_rxn1,rate_rxn2,. . .]]

class `catmap.mappers.mapper_base.`**`MapperBase`**(*reaction_model=None*)
    Bases: *`catmap.ReactionModelWrapper`*

`__getattr__`(*attr*)
    Return the value of the reaction model instance if its there. Otherwise return the instances own value (or none if the instance does not have the attribute defined and the attribute is not private)

`__init__`(*reaction_model=None*)

`__module__ = 'catmap.mappers.mapper_base'`

`__setattr__`(*attr, val*)
    Set attribute for the instance as well as the reaction_model instance

`get_output_map`(*descriptor_ranges, resolution, *args, **kwargs*)

`get_point_output`(*descriptors, *args, **kwargs*)

`process_resolution`(*descriptor_ranges=None, resolution=None*)

catmap.mappers.min_resid_mapper module

class `catmap.mappers.min_resid_mapper.`**`MinResidMapper`**(*reaction_model=None*)
    Bases: *`catmap.mappers.mapper_base.MapperBase`*

    Mapper which uses initial guesses with minimum residual.

`__getattr__`(*attr*)
Return the value of the reaction model instance if its there. Otherwise return the instances own value (or none if the instance does not have the attribute defined and the attribute is not private)

`__init__`(*reaction_model=None*)

`__module__ = 'catmap.mappers.min_resid_mapper'`

`__setattr__`(*attr*, *val*)
Set attribute for the instance as well as the reaction_model instance

`bisect_descriptor_line`(*new_descriptors*, *old_descriptors*, *initial_guess_coverages*)
Find coverages at point new_descriptors given that coverages are initial_guess_coverages at old_descriptors by incrementally halving the distance between points upon failure to converge.

param new_descriptors list of descriptors that fails

type new_descriptors [float]

param old_descriptors list of descriptors that is known to work

type old_descriptors [float]

param inititial_guess_coverages List of best of guess for coverages

type initial_guess_coverages [float]

`get_coverage_map`(*descriptor_ranges=None*, *resolution=None*, *initial_guess_adsorbate_names=None*)
Creates coverage map by computing residuals from nearby points and trying points with lowest residuals first

`get_initial_coverage`(*descriptors*, **args*, ***kwargs*)
Return initial guess for coverages based on Boltzmann weights. The return format is [descriptors, [coverages]] where the list of coverages represents the initial guess for different choices for gas phase-reservoirs that are in equilibrium with the surface coverages.

Parameters

- **`descriptors`** (*`A list of descriptor values, like [5, 5]`*) – [float]
- **`*args`** – see catmap.solver.get_initial_coverages
- **`**kwargs`** – see catmap.solver.get_initial_coverages

`get_initial_coverage_from_map`(*descriptors*, **args*, ***kwargs*)

`get_output_map`(*descriptor_ranges*, *resolution*, **args*, ***kwargs*)

`get_point_coverage`(*descriptors*, **args*, ***kwargs*)
Shortcut to get final coverages at a point.

Parameters

- **`descriptors`** (*`[float]`*) – List of chemical descriptors, like [-.5, -.5]
- **`*args`** – see catmap.solvers.get_coverage
- **`**kwargs`** – see catmap.solver.get_coverage

`get_point_output`(*descriptors*, **args*, ***kwargs*)

`process_resolution`(*descriptor_ranges=None*, *resolution=None*)

Module contents

4.1.4 catmap.parsers package

Submodules

catmap.parsers.parser_base module

class `catmap.parsers.parser_base.`**`ParserBase`**(*reaction_model=None*)

Bases: *`catmap.ReactionModelWrapper`*

`__getattr__`(*attr*)

Return the value of the reaction model instance if its there. Otherwise return the instances own value (or none if the instance does not have the attribute defined and the attribute is not private)

`__init__`(*reaction_model=None*)

Class for 'parsing' information from raw data (databases, spreadsheets, text files, trajectories, etc.) into a structure which is useful to the microkinetic model. This class acts as a base class to be inherited by other parser classes, but it is not functional on its own.

input_file: defines the file path or object to get data from

A functional derived parser class must also contain the methods:

parse(input_file): a function to parse the input_file file/object and return properly formatted data. The parse function should save all necessary attributes to the Parser class. After parsing the parent microkinetic model class will update itself from the Parser attributes.

`__module__ = 'catmap.parsers.parser_base'`

`__setattr__`(*attr*, *val*)

Set attribute for the instance as well as the reaction_model instance

`_baseparse`()

catmap.parsers.table_parser module

class `catmap.parsers.table_parser.`**`TableParser`**(*reaction_model=None*, ***kwargs*)

Bases: *`catmap.parsers.parser_base.ParserBase`*

Parses attributes based on column headers and filters.

Additional functionality may be added by inheriting and defining the parse_{header_name} function where header_name is the column header for the additional variable to be parsed.

`__getattr__`(*attr*)

Return the value of the reaction model instance if its there. Otherwise return the instances own value (or none if the instance does not have the attribute defined and the attribute is not private)

`__init__`(*reaction_model=None*, ***kwargs*)

`__module__ = 'catmap.parsers.table_parser'`

`__setattr__`(*attr*, *val*)

Set attribute for the instance as well as the reaction_model instance

`_baseparse`()

`parse`(***kwargs*)

`parse_coverage`(***kwargs*)

parse_formation_energy(***kwargs*)
Parse in basic info for reaction model

parse_frequencies(***kwargs*)

Module contents

4.1.5 catmap.scalers package

Submodules

catmap.scalers.generalized_linear_scaler module

class catmap.scalers.generalized_linear_scaler.**GeneralizedLinearScaler**(*reaction_model=None*)
Bases: *catmap.scalers.scaler_base.ScalerBase*

TODO

__getattr__(*attr*)
Return the value of the reaction model instance if its there. Otherwise return the instances own value (or none if the instance does not have the attribute defined and the attribute is not private)

__init__(*reaction_model=None*)

__module__ = 'catmap.scalers.generalized_linear_scaler'

__setattr__(*attr, val*)
Set attribute for the instance as well as the reaction_model instance

get_adsorbate_coefficient_matrix()
Calculate coefficients for scaling all adsorbates and transition states using constrained optimization. Store results in self.total_coefficient_dict and return the coefficient matrix for the adsorbates only.

get_coefficient_matrix()

TODO

get_electronic_energies(*descriptors*)

TODO

get_energetics(*descriptors*)

get_enthalpies(*descriptors, **kwargs*)

get_entropies(*descriptors, **kwargs*)

get_formation_energy_interaction_parameters(*descriptors*)

TODO

get_formation_energy_parameters(*descriptors*)

TODO

get_free_energies(*descriptors, **kwargs*)

get_rxn_parameters(*descriptors, *args, **kwargs*)

TODO

get_thermodynamic_energies(***kwargs*)

get_total_enthalpies(*descriptors, **kwargs*)

get_transition_state_coefficient_matrix()

TODO

get_transition_state_scaling_matrix()

TODO

parameterize()

TODO

parse_constraints(*constraint_dict*)
This function converts constraints which are input as a dictionary to lists compatible with the function to obtain scaling coefficients.

set_output_attrs(*descriptors*)
Function to set output information.

summary_text()

TODO

catmap.scalers.null_scaler module

class catmap.scalers.null_scaler.**NullScaler**(*reaction_model=None*)
Bases: *catmap.scalers.scaler_base.ScalerBase*

Scaler which passes descriptor values directly to solver

__getattr__(*attr*)
Return the value of the reaction model instance if its there. Otherwise return the instances own value (or none if the instance does not have the attribute defined and the attribute is not private)

__init__(*reaction_model=None*)
Class for 'scaling' descriptors to free energies of reaction and activation (or other parameters). This class acts as a base class to be inherited by other scaler classes, but is not functional on its own.

This class contains the description of the microkinetic model (adsorbate_names, gas_names, etc.) along with the temperature and gas_pressures. In most cases these will automatically be populated by the parent reaction_model class. The scaler-specific attributes are:

gas_energies: defines the energies of the gas-phase species. This sets the references for the system.

gas_thermo_mode: the mode for obtaining thermal contributions in the gas phase. Default is to use the ideal gas approxmation.

adsorbate_thermo_mode: the mode for obtaining thermal contributions from adsorbed species. Default is to use the harmonic adsorbate approximation.

frequency_dict: a dictionary of vibrational frequencies (in eV) for each gas/adsorbate. Should be of the form frequency_dict[ads] = [freq1, freq2,…]. Needed for ideal gas, harmonic adsorbate, or hindered adsorbate approximations.

A functional derived scaler class must also contain the methods:

get_electronic_energies(descriptors): a function to 'scale' the descriptors to electronic energies. Returns a dictionary of the electronic energies of each species in the model.

get_energetics(descriptors): a function to obtain the reaction energetics from the descriptors. Should return a list of length N (number of elementary reactions): [[E_rxn1,E_a1],[E_rxn2,E_a2],…[E_rxnN,E_aN]]

get_rxn_parameters(descriptors): a function to obtain all necessary reaction parameters from the descriptors. Should return a list of length N (number of elementary reactions): [[param1_rxn1,param2_rxn1. . .]. . . [param1_rxnN,param2_rxnN. . .]]. For a simple model this could be the same as get_energetics, but models accounting for interactions may require more parameters which can be scaled.

`__module__ = 'catmap.scalers.null_scaler'`

`__setattr__`(*attr*, *val*)
Set attribute for the instance as well as the reaction_model instance

`get_electronic_energies`(*descriptors*)

`get_energetics`(*descriptors*)

`get_enthalpies`(*descriptors*, ***kwargs*)

`get_entropies`(*descriptors*, ***kwargs*)

`get_free_energies`(*descriptors*, ***kwargs*)

`get_rxn_parameters`(*descriptors*)

`get_thermodynamic_energies`(***kwargs*)

`get_total_enthalpies`(*descriptors*, ***kwargs*)

`set_output_attrs`(*descriptors*)
Function to set output information.

`summary_text`()

catmap.scalers.scaler_base module

`class catmap.scalers.scaler_base.ScalerBase`(*reaction_model=None*)
Bases: *catmap.ReactionModelWrapper*

`__getattr__`(*attr*)
Return the value of the reaction model instance if its there. Otherwise return the instances own value (or none if the instance does not have the attribute defined and the attribute is not private)

`__init__`(*reaction_model=None*)
Class for 'scaling' descriptors to free energies of reaction and activation (or other parameters). This class acts as a base class to be inherited by other scaler classes, but is not functional on its own.

This class contains the description of the microkinetic model (adsorbate_names, gas_names, etc.) along with the temperature and gas_pressures. In most cases these will automatically be populated by the parent reaction_model class. The scaler-specific attributes are:

gas_energies: defines the energies of the gas-phase species. This sets the references for the system.

gas_thermo_mode: the mode for obtaining thermal contributions in the gas phase. Default is to use the ideal gas approxmation.

adsorbate_thermo_mode: the mode for obtaining thermal contributions from adsorbed species. Default is to use the harmonic adsorbate approximation.

frequency_dict: a dictionary of vibrational frequencies (in eV) for each gas/adsorbate. Should be of the form frequency_dict[ads] = [freq1, freq2,. . .]. Needed for ideal gas, harmonic adsorbate, or hindered adsorbate approximations.

A functional derived scaler class must also contain the methods:

get_electronic_energies(descriptors): a function to 'scale' the descriptors to electronic energies. Returns a dictionary of the electronic energies of each species in the model.

get_energetics(descriptors): a function to obtain the reaction energetics from the descriptors. Should return a list of length N (number of elementary reactions): [[E_rxn1,E_a1],[E_rxn2,E_a2],. . . [E_rxnN,E_aN]]

get_rxn_parameters(descriptors): a function to obtain all necessary reaction parameters from the descriptors. Should return a list of length N (number of elementary reactions): [[param1_rxn1,param2_rxn1. . .]. . . [param1_rxnN,param2_rxnN. . .]]. For a simple model this could be the same as get_energetics, but models accounting for interactions may require more parameters which can be scaled.

`__module__ = 'catmap.scalers.scaler_base'`

`__setattr__`(*attr*, *val*)
Set attribute for the instance as well as the reaction_model instance

`get_electronic_energies`(*descriptors*)

`get_energetics`(*descriptors*)

`get_enthalpies`(*descriptors*, ***kwargs*)

`get_entropies`(*descriptors*, ***kwargs*)

`get_free_energies`(*descriptors*, ***kwargs*)

`get_rxn_parameters`(*descriptors*)

`get_thermodynamic_energies`(***kwargs*)

`get_total_enthalpies`(*descriptors*, ***kwargs*)

`set_output_attrs`(*descriptors*)
Function to set output information.

`summary_text`()

catmap.scalers.thermodynamic_scaler module

class `catmap.scalers.thermodynamic_scaler.`**`ThermodynamicScaler`**(*reaction_model=None*)
Bases: `catmap.scalers.scaler_base.ScalerBase`

Scaler which uses temperature/pressure/potential as descriptors and treates energetics as a constant

`__getattr__`(*attr*)
Return the value of the reaction model instance if its there. Otherwise return the instances own value (or none if the instance does not have the attribute defined and the attribute is not private)

`__init__`(*reaction_model=None*)

`__module__ = 'catmap.scalers.thermodynamic_scaler'`

`__setattr__`(*attr*, *val*)
Set attribute for the instance as well as the reaction_model instance

`get_electronic_energies`(*descriptors*)

`get_energetics`(*descriptors*)

`get_enthalpies`(*descriptors*, ***kwargs*)

`get_entropies`(*descriptors*, ***kwargs*)

get_formation_energy_interaction_parameters(*descriptors*)

get_formation_energy_parameters(*descriptors*)

get_free_energies(*descriptors*, ***kwargs*)

get_rxn_parameters(*descriptors*, **args*, ***kwargs*)

get_thermodynamic_energies(*descriptors*, ***kwargs*)

get_total_enthalpies(*descriptors*, ***kwargs*)

set_output_attrs(*descriptors*)
Function to set output information.

summary_text()

Module contents

4.1.6 catmap.solvers package

Submodules

catmap.solvers.integrated_rate_control_solver module

class catmap.solvers.integrated_rate_control_solver.**IntegratedRateControlSolver**(*reaction_model*
Bases: *catmap.solvers.solver_base.SolverBase*

Class for estimating rates based on the degree of rate control screening method {citation after published}

__getattr__(*attr*)
Return the value of the reaction model instance if its there. Otherwise return the instances own value (or none if the instance does not have the attribute defined and the attribute is not private)

__init__(*reaction_model=None*)

__module__ = 'catmap.solvers.integrated_rate_control_solver'

__setattr__(*attr*, *val*)
Set attribute for the instance as well as the reaction_model instance

compile()

get_integrated_DRC_rate(*params*, **args*, ***kwargs*)

set_output_attrs(*params*)

catmap.solvers.mean_field_solver module

class catmap.solvers.mean_field_solver.**MeanFieldSolver**(*reaction_model=None*)
Bases: *catmap.solvers.solver_base.SolverBase*

Class for handling mean-field type kinetic models. Can be sub-classed to get functionality for steady-state solutions, sabatier solutions, etc.

__getattr__(*attr*)
Return the value of the reaction model instance if its there. Otherwise return the instances own value (or none if the instance does not have the attribute defined and the attribute is not private)

__init__(*reaction_model=None*)

__module__ = 'catmap.solvers.mean_field_solver'

`__setattr__`(*attr*, *val*)
Set attribute for the instance as well as the reaction_model instance

`get_apparent_activation_energy`(*rxn_parameters*, *epsilon=1e-10*)
returns apparent Arrhenius activation energies (in units of R) for production/consumption of each gas phase species. Calculated as E_app = T^2(dlnr_+/dT)=(T^2/r_+)(dr_+/dT), where r+ is the TOF :param rxn_parameters: reaction paramenters, see solver-base :param epsilon: degree of pertubation in temperature :type epsilon: float, optional

`get_directional_rates`(*rxn_parameters*)
get the exchange current density of a certain map entry on all elementary rxns for a given direction

type: direction: str (‘kf’ or ‘kr’)

`get_elem_ec`(*rxn_num*, *rxn_parameters*, *direction*)
return the ec on a certain coverage_map entry of a certain elementary step based on rxn_number and direction given

type: rxn_num: float direction: str (‘kf’ or ‘kr’)

`get_empty_site_cvgs`()
take the coverages at a certain coverage_map entry and return the dict of all the empty-sites coverages i.e. dict[site_name] = coverage

type: coverages: list

`get_interacting_energies`(*rxn_parameters*)
return the integral energy under high coverage with interactions :param rxn_parameters: reaction parameters, see solver-base

`get_rate`(*rxn_parameters*, *coverages=None*, *verify_coverages=True*, ***coverage_kwargs*)

`get_rate_control`(*rxn_parameters*)
return list of degree of rate control for each reaction Ref: Stegelmann et al., DOI: 10.1021/ja9000097 :param rxn_parameters: reaction parameters, see solver-base

`get_rxn_order`(*rxn_parameters*, *epsilon=1e-10*)
return the reaction orders for the reactants :param rxn_parameters: reaction parameters, see solver-base :param epsilon: degree of perturbation in pressure :type epsilon: float, optional

`get_rxn_rates`(*coverages*, *rate_constants*)
returns list of reaction rate for each elementary reaction based on reaction constants & coverage .. todo:: coverages, rate_constants

`get_selectivity`(*rxn_parameters*, *weights=None*)
return list of selectivity of each reaction :param rxn_parameters: reaction parameters, see solver-base :param weights: weights for each species. Defaults to 1 for all species

`get_selectivity_control`(*rxn_parameters*)
return the list of degree of selectivity control for each rxn :param rxn_parameters: reaction parameters, see solver-base

`get_turnover_frequency`(*rxn_parameters*, *rates=None*, *verify_coverages=True*)
return list of turnover frequencies of all the gas-phase species :param rates: list of rates of each rxn ineq_cons = {‘type’: ‘ineq’,

> ‘fun’ : lambda x: x, ‘jac’ : lambda x: np.eye(*np.shape(x))}

Parameters

- **`verify_coverages`** (`bool, optional`) – verify that the species has a certain value for the coverage

- **rxn_parameters** – reaction parameters, see solver-base

jacobian_equations(*adsorbate_interactions=True*)
Composes analytical expressions for the Jacobian matrix. Assumes:

kf is defined as a list of forward rate-constants kr is defined as a list of reverse rate-constants theta is defined as a list of coverages p is defined as a list of pressures

If the rate constants depend on coverage, use adsorbate_interactions = True. Assumes:

kB is defined as Boltzmann's constant T is defined as the temperature dEf is defined as a list of lists where dEf[i][j] is the

derivative of forward activation free energy i wrt coverage j

dEr is defined as a list of lists where dEr[i][j] is the derivative of reverse activation free energy i wrt coverage j

Param adsorbate_interactions: tell if need to include interactions

Type adsorbate_interactions: bool, optional

rate_equation_term(*species_list*, *rate_constant_string*, *d_wrt=None*)
Function to compose a term in the rate equation - e.g. kf[1]*theta[0]*p[0] :param species_list: list of species in rate equations :type species_list: list

rate_equations()
Compose analytical expressions for the reaction rates and change of surface species wrt time (dc/dt). Assumes:

kf is defined as a list of forward rate-constants kr is defined as a list of reverse rate-constants theta is defined as a list of coverages p is defined as a list of pressures

reaction_energy_equations(*adsorbate_interactions=True*)
Composes a list of analytical expressions which give the reaction and activation energies for elementary steps. Note that while this is useful primarily for models with adsorbate-interactions (otherwise these energetics can easily be obtained by the reaction model itself), they are technically valid for all mean-field models. Assumes:

Gf is a list of formation energies ordered as adsorbate_names+transition_state_names

If model includes adsorbate interactions then use adsorbate_interactions = True to include dEa/dtheta in the output. Assumes:

dGs is a matrix/array of derivatives of free energies wrt coverages such that dGs[:,i] is a vector of derivatives of the free energy of species i wrt each coverage ordered as adsorbate_names

Param adsorbate_interaction: specify whether or not to include interactions

Type adsorbate_interaction: bool, optional

set_output_attrs(*rxn_parameters*)

Parameters **rxn_parameters** (*list*) – Reaction parameters.

site_string_list()
Function to compose an analytic expression for the coverage of empty sites

substitutions_dict()
Dictionary of substitutions needed for static compiled functions

summary_text()
Stub for producing solver summary.

catmap.solvers.solver_base module

class catmap.solvers.solver_base.**NewtonRoot**(*f*, *x0*, *matrix*, *mpfloat*, *Axb_solver*, ***kwargs*)
Hacked from MDNewton in mpmath/calculus/optimization.py in order to allow for constraints on the solution.

Find the root of a vector function numerically using Newton's method.

f is a vector function representing a nonlinear equation system.

x0 is the starting point close to the root.

J is a function returning the Jacobian matrix for a point.

Supports overdetermined systems.

Use the 'norm' keyword to specify which norm to use. Defaults to max-norm. The function to calculate the Jacobian matrix can be given using the keyword 'J'. Otherwise it will be calculated numerically.

Please note that this method converges only locally. Especially for high- dimensional systems it is not trivial to find a good starting point being close enough to the root.

It is recommended to use a faster, low-precision solver from SciPy [1] or OpenOpt [2] to get an initial guess. Afterwards you can use this method for root-polishing to any precision.

[1] http://scipy.org

[2] http://openopt.org

__init__(*f*, *x0*, *matrix*, *mpfloat*, *Axb_solver*, ***kwargs*)

__iter__()

__module__ = 'catmap.solvers.solver_base'

maxsteps = 10

class catmap.solvers.solver_base.**SolverBase**(*reaction_model=None*)
Bases: *catmap.ReactionModelWrapper*

__getattr__(*attr*)
Return the value of the reaction model instance if its there. Otherwise return the instances own value (or none if the instance does not have the attribute defined and the attribute is not private)

__init__(*reaction_model=None*)
Class for 'solving' for equilibrium coverages and rates as a function of reaction parameters. This class acts as a base class to be inherited by other solver classes, but is not functional on its own.

rxn_parameters: list of necessary parameters to solve the kinetic system. This will usually be populated by the scaler.

A functional derived solver class must also contain the methods:

get_coverage(): a function which returns coverages for each adsorbate as a list [cvg_ads1,cvg_ads2,. . .]

get_rate(): a function which returns steady-state reaction rates for each elementary step as a list [rate_rxn1,rate_rxn2,. . .]

get_residual(): a function for computing the norm of the residual. This is the condition which will be minimized to reach steady-state.

compile(): a function to set-up/compile the solver.

`__module__ = 'catmap.solvers.solver_base'`

`__setattr__`(*attr*, *val*)
Set attribute for the instance as well as the reaction_model instance

`set_output_attrs`(*rxn_parameters*)

Parameters **`rxn_parameters`** (*`list`*) – Reaction parameters.

catmap.solvers.steady_state_solver module

`class` `catmap.solvers.steady_state_solver.`**`SteadyStateSolver`**(*reaction_model=None*)
Bases: *`catmap.solvers.mean_field_solver.MeanFieldSolver`*

`__getattr__`(*attr*)
Return the value of the reaction model instance if its there. Otherwise return the instances own value (or none if the instance does not have the attribute defined and the attribute is not private)

`__init__`(*reaction_model=None*)

`__module__ = 'catmap.solvers.steady_state_solver'`

`__setattr__`(*attr*, *val*)
Set attribute for the instance as well as the reaction_model instance

`bisect_interaction_strength`(*rxn_parameters*, *valid_strength*, *valid_coverages*, *target_strength*, *max_bisections*, *findrootArgs={}*)

TODO

`compile`()

TODO

`constrain_coverages`(*cvgs*)

TODO

`get_apparent_activation_energy`(*rxn_parameters*, *epsilon=1e-10*)
returns apparent Arrhenius activation energies (in units of R) for production/consumption of each gas phase species. Calculated as E_app = T^2(dlnr_+/dT)=(T^2/r_+)(dr_+/dT), where r+ is the TOF :param rxn_parameters: reaction paramenters, see solver-base :param epsilon: degree of pertubation in temperature :type epsilon: float, optional

`get_coverage`(*rxn_parameters*, *c0=None*, *findrootArgs={}*)
Return coverages for given reaction parameters and coverage constraints.

Parameters

- **`rxn_parameters`** (*`[float]`*) – Sequence of rxn_parameters
- **`c0`** (**TODO**) – Coverage constraints.
- **`findrootArgs`** – *deprecated*

`get_directional_rates`(*rxn_parameters*)
get the exchange current density of a certain map entry on all elementary rxns for a given direction

type: direction: str (‘kf’ or ‘kr’)

`get_elem_ec`(*rxn_num*, *rxn_parameters*, *direction*)
return the ec on a certain coverage_map entry of a certain elementary step based on rxn_number and direction given

type: rxn_num: float direction: str (‘kf’ or ‘kr’)

get_empty_site_cvgs()
take the coverages at a certain coverage_map entry and return the dict of all the empty-sites coverages i.e. dict[site_name] = coverage

type: coverages: list

get_ideal_coverages(*rxn_parameters*, *c0=None*, *refresh_rate_constants=True*, *findrootArgs={}*)
Return

TODO

get_initial_coverage(*rxn_parameters*)
Return coverages based on probabilties according to the Boltzmann distribution and the adsorption energies for a given sequence of rxn_parameters.

Parameters **rxn_parameters** – Sequence of reaction parameters

get_interacting_coverages(*rxn_parameters*, *c0=None*, *interaction_strength=1.0*, *findrootArgs={}*)

TODO

get_interacting_energies(*rxn_parameters*)
return the integral energy under high coverage with interactions :param rxn_parameters: reaction parameters, see solver-base

get_rate(*rxn_parameters*, *coverages=None*, *verify_coverages=True*, ***coverage_kwargs*)

get_rate_constants(*rxn_parameters*, *coverages*)
Return rate constants for given sequence of reaction parameters and coverages.

Parameters

- **rxn_parameters** (*[float]*) – Sequence of reaction parameters.
- **coverages** (*[float]*) – Sequence of coverages.

get_rate_control(*rxn_parameters*)
return list of degree of rate control for each reaction Ref: Stegelmann et al., DOI: 10.1021/ja9000097 :param rxn_parameters: reaction parameters, see solver-base

get_residual(*coverages*, *validate_coverages=True*, *refresh_rate_constants=True*)

TODO

get_rxn_order(*rxn_parameters*, *epsilon=1e-10*)
return the reaction orders for the reactants :param rxn_parameters: reaction parameters, see solver-base :param epsilon: degree of perturbation in pressure :type epsilon: float, optional

get_rxn_rates(*coverages*, *rate_constants*)
returns list of reaction rate for each elementary reaction based on reaction constants & coverage .. todo:: coverages, rate_constants

get_selectivity(*rxn_parameters*, *weights=None*)
return list of selectivity of each reaction :param rxn_parameters: reaction parameters, see solver-base :param weights: weights for each species. Defaults to 1 for all species

get_selectivity_control(*rxn_parameters*)
return the list of degree of selectivity control for each rxn :param rxn_parameters: reaction parameters, see solver-base

get_steady_state_coverage(*rxn_parameters*, *steady_state_fn*, *jacobian_fn*, *c0=None*, *findrootArgs={}*)
Return steady-state coverages using catmap.solvers.solver_base.NewtonRoot .

Parameters

- **rxn_parameters** (*[float]*) – Sequence of reaction parameters.
- **steady_state_fn** (**TODO**) – **TODO**
- **jacobian_fn** (**TODO**) – **TODO**
- **c0** (**TODO**) – Coverage constraints
- **findrootArgs** – *deprecated*

get_turnover_frequency(*rxn_parameters*, *rates=None*, *verify_coverages=True*)

return list of turnover frequencies of all the gas-phase species :param rates: list of rates of each rxn ineq_cons = {'type': 'ineq',

'fun' : lambda x: x, 'jac' : lambda x: np.eye(*np.shape(x))}

Parameters

- **verify_coverages** (*bool, optional*) – verify that the species has a certain value for the coverage
- **rxn_parameters** – reaction parameters, see solver-base

ideal_steady_state_function(*coverages*)

TODO

ideal_steady_state_jacobian(*coverages*)

TODO

interacting_steady_state_function(*coverages*)

TODO

interacting_steady_state_jacobian(*coverages*)

TODO

jacobian_equations(*adsorbate_interactions=True*)

Composes analytical expressions for the Jacobian matrix. Assumes:

kf is defined as a list of forward rate-constants kr is defined as a list of reverse rate-constants theta is defined as a list of coverages p is defined as a list of pressures

If the rate constants depend on coverage, use adsorbate_interactions = True. Assumes:

kB is defined as Boltzmann's constant T is defined as the temperature dEf is defined as a list of lists where dEf[i][j] is the

derivative of forward activation free energy i wrt coverage j

dEr is defined as a list of lists where dEr[i][j] is the derivative of reverse activation free energy i wrt coverage j

Param adsorbate_interactions: tell if need to include interactions

Type adsorbate_interactions: bool, optional

optimize_analytical_function(*func_name*, *func_string*, *insertion_line*, *indention_level*, **test_args*)

Replace some common multiplication terms to speed up functions.

rate_equation_term(*species_list*, *rate_constant_string*, *d_wrt=None*)
Function to compose a term in the rate equation - e.g. kf[1]*theta[0]*p[0] :param species_list: list of species in rate equations :type species_list: list

rate_equations()
Compose analytical expressions for the reaction rates and change of surface species wrt time (dc/dt). Assumes:

kf is defined as a list of forward rate-constants kr is defined as a list of reverse rate-constants theta is defined as a list of coverages p is defined as a list of pressures

reaction_energy_equations(*adsorbate_interactions=True*)
Composes a list of analytical expressions which give the reaction and activation energies for elementary steps. Note that while this is useful primarily for models with adsorbate-interactions (otherwise these energetics can easily be obtained by the reaction model itself), they are technically valid for all mean-field models. Assumes:

Gf is a list of formation energies ordered as adsorbate_names+transition_state_names

If model includes adsorbate interactions then use adsorbate_interactions = True to include dEa/dtheta in the output. Assumes:

dGs is a matrix/array of derivatives of free energies wrt coverages such that dGs[:,i] is a vector of derivatives of the free energy of species i wrt each coverage ordered as adsorbate_names

Param adsorbate_interaction: specify whether or not to include interactions

Type adsorbate_interaction: bool, optional

set_output_attrs(*rxn_parameters*)

Parameters **rxn_parameters** (*list*) – Reaction parameters.

site_string_list()
Function to compose an analytic expression for the coverage of empty sites

substitutions_dict()
Dictionary of substitutions needed for static compiled functions

summary_text()
Stub for producing solver summary.

Module contents

4.1.7 catmap.thermodynamics package

Submodules

catmap.thermodynamics.enthalpy_entropy module

class catmap.thermodynamics.enthalpy_entropy.**ThermoCorrections**(*reaction_model=None*)
Bases: *catmap.ReactionModelWrapper*

Class for including thermodynamic corrections.

The function “get_thermodynamic_corrections” automatically does all the work assuming the correct functions are in place.

thermodynamic_corrections: List of fundamentally different types of corrections which could be included. Defaults are gas and adsorbate but other possibilities might be interface, electrochemical, etc.

thermodynamic_variables: List of variables which define a thermodynamic state. If these attributes of the underlying reaction model do not change then the thermodynamic corrections will not be recalculated in order to save time.

To add a new correction type (called custom_correction):

1) **Define the function which performs the correction as an attribute.** Assume the function is called "simple_custom_correction".
2) Place the "custom_correction" in the "thermodynamic_corrections" list
3) **Place any variables which the custom correction depends on in** the thermodynamic_variables list
4) **Set the "custom_correction_thermo_mode" attribute of the** underlying reaction model to "simple_custom_correction"

If these steps are followed then the correction should automatically be included in all calculations.

`__getattr__`(*attr*)
Return the value of the reaction model instance if its there. Otherwise return the instances own value (or none if the instance does not have the attribute defined and the attribute is not private)

`__init__`(*reaction_model=None*)

`__module__ = 'catmap.thermodynamics.enthalpy_entropy'`

`__setattr__`(*attr*, *val*)
Set attribute for the instance as well as the reaction_model instance

`_bar2Pa = 100000.0`

`_get_echem_corrections`(*correction_dict*)
Perform the thermodynamic corrections relevant to electrochemistry but are not specific to any particular mode.

`_kJmol2eV = 0.01036427`

`_shomate_eq`(*params*, *temperature=[]*)

`approach_to_equilibrium_pressure`()
Set product pressures based on approach to equilibrium. Requires the following attributes to be set: global_reactions - a list of global reactions in the same syntax as elementary expressions,

with each one followed by its respective approach to equilibrium.

pressure_mode - must be set to 'approach_to_equilibrium' Note that this function is not well-tested and should be used with caution.

`average_transition_state`(*thermo_dict*, *transition_state_list=[]*, *thermo_vars=[]*)
Return transition state thermochemical corrections as average of IS and FS corrections

`boltzmann_coverages`(*energy_dict*)
Return coverages based on Boltzmann distribution

`concentration_pressure`()

`estimate_hbond_corr`()
Generate hydrogen bonding corrections given a formula and estimations for various functional groups used in Peterson(2010) - valid mostly for Pt(111) This is a very simplistic function. If you need more advanced descriptions of hydrogen bonding, consider setting your own hbond_dict.

`fixed_enthalpy_entropy_adsorbate`()
Return free energy corrections based on input enthalpy, entropy, ZPE

fixed_enthalpy_entropy_gas(*gas_names=None*)
Calculate free energy corrections based on input enthalpy, entropy, ZPE

fixed_entropy_gas(*include_ZPE=True*)
Add entropy based on fixed_entropy_dict (entropy contribution to free energy assumed linear with temperature) and ZPE

frozen_adsorbate()
Neglect all zero point, enthalpy, entropy corrections to adsorbate energy.

frozen_fixed_entropy_gas()
Do not add ZPE, calculate fixed entropy correction.

frozen_gas()
Neglect all thermal contributions, including the zero point energy.

generate_echem_TS_energies()
Give real energies to the fake echem transition states

get_frequency_cutoff(*kB_multiplier*, *temperature=None*)

get_pressure_equilibrium(*xguess=None*, *ftol=1e-05*)

get_rxn_index_from_TS(*TS*)
Take in the name of a transition state and return the reaction index of the elementary rxn from which it belongs

get_thermodynamic_corrections(***kwargs*)
Calculate all "thermodynamic" corrections beyond the energies in the input file. This master function will call sub-functions depending on the "thermo mode" of each class of species

harmonic_adsorbate()
Calculate the thermal correction to the free energy of an adsorbate in the harmonic approximation using the HarmonicThermo class in ase.thermochemistry.

adsorbate_names = the chemical formulas of the adsorbates of interest. freq_dict = dictionary of vibrational frequencies for each adsorbate of

> interest. Vibrational frequencies should be in eV. The dictionary should be of the form freq_dict[ads_name] = [freq1, freq2, . . .]

hbond_electrochemical()
Update simple_electrochemical with hbonding corrections as if they were on Pt(111)

hbond_with_estimates_electrochemical()
Add hbond corrections to transition states involving pe and ele (coupled proton-electron transfers and electron transfers)

hindered_adsorbate()
Calculate the thermal correction to the free energy of an adsorbate in the hindered translator and hindered rotor model using the HinderedThermo class in ase.thermochemistry along with the molecular structures in ase.data.molecules. Requires ase version 3.12.0 or greater.

adsorbate_names = the chemical formulas of the adsorbates of interest. freq_dict = dictionary of vibrational frequencies for each adsorbate of

> interest. Vibrational frequencies should be in eV. The dictionary should be of the form freq_dict[ads_name] = [freq1, freq2, . . .]

hindered_ads_params = dictionary containing for each adsorbate

> **[0] = translational energy barrier in eV (barrier for the** adsorbate to diffuse on the surface)

[1] = rotational energy barrier in eV (barrier for the adsorbate to rotate about an axis perpendicular to the surface)

[2] = surface site density in cm^-2 [3] = number of equivalent minima in full adsorbate rotation [4] = mass of the adsorbate in amu (can be unspecified by putting

None, in which case mass will attempt to be calculated from the ase atoms class)

[5] = reduced moment of inertia of the adsorbate in amu*Ang^-2 (can be unspecified by putting None, in which case inertia will attempt to be calculated from the ase atoms class)

[6] = symmetry number of the adsorbate (number of symmetric arms of the adsorbate which depends upon how it is bound to the surface. For example, propane bound through its end carbon has a symmetry number of 1 but propane bound through its middle carbon has a symmetry number of 2. For single atom adsorbates such as O* the symmetry number is 1.)

The dictionary should be of the form hindered_ads_params[ads_name] = [barrierT, barrierR, site_density, rotational_minima, mass, inertia, symmetry_number]

atoms_dict = dictionary of ase atoms objects to use for calculating mass and rotational inertia. If none is specified then the function will look in ase.data.molecules. Can be omitted if both mass and rotational inertia are specified in hindered_ads_params.

`homogeneous_field`()

Update simple_electrochemical with field corrections for adsorbates that respond to a field

`ideal_gas`()

Calculate the thermal correction to the free energy of an ideal gas using the IdealGasThermo class in ase.thermochemistry along with the molecular structures in ase.data.molecules.

gas_names = the chemical formulas of the gasses of interest (usually ending in _g to denote that they are in the gas phase).

freq_dict = dictionary of vibrational frequencies for each gas of interest. Vibrational frequencies should be in eV. The dictionary should be of the form freq_dict[gas_name] = [freq1, freq2, . . .]

ideal_gas_params = dictionary of the symmetry number, geometry keyword, and spin of the gas. If no dictionary is specified then the function will attempt to look the gas up in the hard-coded gas_params dictionary. The dictionary should be of the form ideal_gas_params[gas_name] = [symmetry_number, geometry, spin]

atoms_dict = dictionary of ase atoms objects to use for calculating rotational contributions. If none is specified then the function will look in ase.data.molecules.

`local_field_electrochemical`()

Obtains corrections to thermo_dict in the presence of ions hey you need to specify these things: model.Upzc (float) model.CH (float) model.field_site_name model.unfield_site_name and DO NOT specify beta

`set_affine_pressure_equilibrium`(*alpha*, *x0=[]*)

defines gas_pressure as an affine combination between the actual pressure and that of equilibrium by an alpha factor

`set_equilibrated`()

Set reactants/products as their equilibrium composition a priori such that close-to-equilibrium species TOF do not overshadow thos of species that are far from it. It will look for *.equilibrated* parameter. This function assume ‘static’ pressure, such the extent of reactions toward equilibrium would not lead to significant change in the systems’ total pressure.

shomate_adsorbate()
Calculate the thermal correction to the free energy of an adsorbate using pre-fitted shomate parameters.

shomate_gas()
Calculate free energy corrections using Shomate equation

simple_electrochemical()
Calculate electrochemical (potential) corrections to free energy. Transition state energies are corrected by a beta*voltage term.

static_pressure()

summary_text()

zero_point_adsorbate()
Add zero point energy correction to adsorbate energy.

zero_point_gas()
Add zero point energy correction to gasses.

catmap.thermodynamics.enthalpy_entropy.**fit_shomate**(*Ts*, *Cps*, *Hs*, *Ss*, *params0=[]*, *plot_file=None*)
This regression functionality has been updated from a non-linear version to a linearized one which does not need initial guesses (params0). params0 was kept as parameter to the function for backwards compatibiliy. It should be following deprecated.

catmap.thermodynamics.enthalpy_entropy.**harmonic_to_shomate**(*frequencies*, *Tmin*, *Tmax*, *resolution*)
Generate Shomate parameters as of frequency data by using *fit_shomate*.

catmap.thermodynamics.first_order_interactions module

class catmap.thermodynamics.first_order_interactions.**FirstOrderInteractions**(*reaction_model=Non*
Bases: *catmap.ReactionModelWrapper*

Class for implementing 'first-order adsorbate interaction model. Should be sub-classed by scaler.

__getattr__(*attr*)
Return the value of the reaction model instance if its there. Otherwise return the instances own value (or none if the instance does not have the attribute defined and the attribute is not private)

__init__(*reaction_model=None*)

__module__ = 'catmap.thermodynamics.first_order_interactions'

__setattr__(*attr*, *val*)
Set attribute for the instance as well as the reaction_model instance

error_norm(*diff_err*, *int_err*)

fit()

fit_interaction_parameter(*theta_list*, *E_diffs*, *E_ints*, *param_name*, *surf_name*)

fit_old()

get_TS_weight_matrix(*weight*)
Helper function to get 'weights' of how to distribute TS-cross interactions between IS/FS. Should not be called externally.

get_energy_error(*epsilon_ij*, *theta*, *Ediff*, *Eint*, *parameter_name*, *surface_name*)

get_interaction_info()

get_interaction_matrix(*descriptors*)

get_interaction_scaling_matrix()

get_interaction_transition_state_scaling_matrix()

static linear_response(**args*, ***kwargs*)

parameterize_interactions()

params_to_matrix(*param_vector*)

static piecewise_linear_response(**args*, ***kwargs*)

required_interaction_parameters(*cvg*)

static smooth_piecewise_linear_response(**args*, ***kwargs*)

catmap.thermodynamics.second_order_interactions module

class catmap.thermodynamics.second_order_interactions.**SecondOrderInteractions**(*reaction_model=N*

Bases: *catmap.thermodynamics.first_order_interactions.FirstOrderInteractions*, *catmap.ReactionModelWrapper*

Class for implementing 'first-order adsorbate interaction model. Should be sub-classed by scaler.

__getattr__(*attr*)
Return the value of the reaction model instance if its there. Otherwise return the instances own value (or none if the instance does not have the attribute defined and the attribute is not private)

__init__(*reaction_model=None*)

__module__ = 'catmap.thermodynamics.second_order_interactions'

__setattr__(*attr*, *val*)
Set attribute for the instance as well as the reaction_model instance

error_norm(*diff_err*, *int_err*)

fit()

fit_interaction_parameter(*theta_list*, *E_diffs*, *E_ints*, *param_name*, *surf_name*)

fit_old()

get_TS_weight_matrix(*weight*)
Helper function to get 'weights' of how to distribute TS-cross interactions between IS/FS. Should not be called externally.

get_energy_error(*epsilon_ij*, *theta*, *Ediff*, *Eint*, *parameter_name*, *surface_name*)

get_interaction_info()

get_interaction_matrix(*descriptors*)

get_interaction_scaling_matrix()

get_interaction_transition_state_scaling_matrix()

static linear_response(**args*, ***kwargs*)

static offset_smooth_piecewise_linear_response(**args*, ***kwargs*)

parameterize_interactions()

params_to_matrix(*param_vector*)

static piecewise_linear_response(**args*, ***kwargs*)

required_interaction_parameters(*cvg*)

static smooth_piecewise_linear_response(**args*, ***kwargs*)

Module contents

4.2 Submodules

4.3 catmap.model module

class catmap.model.**ReactionModel**(***kwargs*)

The central object that defines a microkinetic model consisting of:

- active sites
- species
- possible reaction steps
- rate constant expressions
- descriptors and descriptor ranges
- data files for energies
- external parameters (temperature, pressures)
- other more technical settings related to the solver and mapper

__init__(***kwargs*)

Class for managing microkinetic models.

Parameters setup_file (*str*) – Specify <mkm-file> from which to load model.

__module__ = 'catmap.model'

_header(*exclude_outputs=[]*, *re_parse=False*)

Create a string which acts as a header for the log file. The header string ensures that the logfile can be opened interactively by Python by importing necessary libraries and automatically reading in the data_file.

Parameters

- **exclude_outputs** (*[str]*) – Attribute names of ReactionModel to exclude in the log file. Optional parameter, default is [].
- **re_parse** (*bool*) – Determines whether or not the parser should be specified in the log file. If a parser is included in the log file then a ReactionModel instantiated using that log file as a setup_file argument will attempt to re-parse values from the input_file and setup_file. Optional parameter, default is False.

_token()

Create a 'token' which uniquely identifies the model based on the user-input parameters. Two models with identical tokens should have identical solutions, although this is not guaranteed if an expert user changes some private attributes.

adsorption_to_reaction_energies(*free_energy_dict*)

Convert adsorption formation energies to reaction energies/barriers.

Parameters **free_energy_dict** (*dict*) – Dictionary containing free energies for each species in the reaction network.

static array_to_map(*array*, *descriptor_ranges*, *resolution*)
Convert numpy array object into CatMAP "map" data structure.

Parameters

- **array** (*numpy.array*) – Numpy array of size (resolution x resolution) corresponding to grid spanning descriptor_ranges.
- **descriptor_ranges** (*[[float]]*) – Minimum and maximum values of descriptor range for each dimension included in array.
- **resolution** (*int*) – Resolution at which the descriptor ranges are sampled.

compatibility_check()
Check that the reaction model has all required attributes. Required attributes can be specified in self._required.

descriptor_space_analysis()
Use mapper to create map/volcano-plot of rates/coverages.

expression_string_to_list(*eq*)

generate_echem_TS()
generates fake transition state species from self.echem_transition_state_names and populates self.species_definitions with them.

generate_echem_species_definitions()
Generates proper species_definitions entries for ele_g, H_g, or OH_g.

generate_static_functions()
Dynamically compile static functions

get_rxn_energy(*rxn*, *energy_dict*)
Calculate reaction energy given the energies of all species.

Parameters

- **rxn** (*[[str]]*) – Reaction in CatMAP form:
  - [[IS],[TS],[FS]] for activated reaction
  - [[IS],[FS]] for non-activated reaction where IS,TS,FS correspond to the names of the species in the initial/transition/final states respectively.
- **energy_dict** (*dict*) – Dictionary of energies for all species. Keys should be species names and values should be energies.

get_state_energy(*rxn_state*, *energy_dict*)
Calculate energy of a "reaction state" (list of species) given the energies of all species.

Parameters

- **rxn_state** – List of intermediate species (must be defined in species_definitions)
- **energy_dict** (*dict*) – Dictionary of energies for all species. Keys should be species names and values should be energies.

load(*setup_file*)
Load a 'setup file' by importing it and assigning all local variables as attributes of the kinetic model. Special attributes mapper, parser, scaler, solver will attempt to convert strings to modules.

`load_data_file`(*overwrite=False*)
Load in output data from external files.

`log`(*event*, ***kwargs*)
Add an event to the log file. This assumes that the template for the event has been specified in the _log_strings attribute of the class that calls the log() function.

Parameters

- **`event`** (`str`) – A key that defines the event to be logged. The _log_strings attribute of the subclass which calls log() should be a dictionary where *event* is a key and the value is a template string. The template string can contain the following variables which will auto-populate:
  - pt - the current point in descriptor space
  - priority - defaults to 0

  In addition, the template may contain other variables which can be passed in as keyword arguments. The following are special arguments:

  *n_iter - the iteration number will be appended to the event title

  The event title should be of the form routinename_status, where routinename is the name of the routine/algorithm and the status is succeess/failure/evaluation/etc.
- **`kwargs`** (`keyword arguments`) – Keyword arguments can be specified and will be passed into the template retrieved from _log_strings['event']

`make_standalone`(*standalone_script='stand_alone.py'*)
Create a stand alone script containing the current model.

Parameters **`standalone_script`** (`str`) – The name of the file where the standalone script is created [stand_alone.py].

static `map_to_array`(*mapp*, *descriptor_ranges*, *resolution*, *log_interpolate=False*, *minval=None*, *maxval=None*)
Convert into CatMAP "map" data structure into numpy array. The "map" will be interpolated onto a regular grid.

Parameters

- **`mapp`** (`CatMAP map (see MapperBase)`) – CatMAP "map" structured lists of descriptor points and corresponding values.
- **`descriptor_ranges`** (`[[float]]`) – Minimum and maximum values of descriptor range for each dimension included in array.
- **`resolution`** (`int`) – Resolution at which the descriptor ranges are sampled.
- **`log_interpolate`** (`bool, optional`) – Take logarithm of values before interpolation. Defaults to False.
- **`minval`** (`float`) – Replace any values less than minval with minval. None implies no cutoff. Defaults to None.
- **`maxval`** (`float`) – Replace any values greater than maxval with maxval. None implies no cutoff. Defaults to None.

`model_summary`(*summary_file='summary.tex'*)
Write summary of model into TeX file.

Parameters **`summary_file`** (`str`) – Filename where TeX summary of model is written.

multi_point_analysis()
Analyze the output at a list of points. Points should be specified as a list in the descriptor_values attribute.

nearest_mapped_point(*mapp*, *point*)
Get the point in the map nearest to the point supplied

parse(**args*, ***kwargs*)
Read in all the information from the input_file. Alias to parser.parse.

parse_elementary_rxns(*equations*)
Convert elementary reaction strings into structured elementary reaction lists.

Parameters **equations** – List of reaction equation strings. For non-activated reactions (e.g. no activation barrier) the strings should follow a syntax like:

- A_s + B_q <-> C_s + D_q
- A_s + B_q -> C_s + D_q

while an activated reaction should follow a syntax like:

- A_s + B_q <-> A-B_s + *_q -> AB_s + *_q

where A,B,C,D are names of chemical species, A-B is the name of a transition-state, and s,q are names of different site types.

:type equations:[str]

static print_point(*descriptors*, *n=2*)
Pretty-print a set of descriptor values.

Parameters

- **descriptors** (*[float]*) – List of descriptor values [d1,d2,…] where d1,d2,… are floats corresponding to coordinates in descriptor space.
- **n** (*int*) – Number of decimals to print out. Optional parameter, default is 2.

print_rxn(*rxn*, *mode='latex'*, *include_TS=True*, *print_out=False*)
Print a structured elementary step and print it as plain text or latex.

Parameters

- **rxn** (*[[str]]*) – Elementary step list of the form [[IS1,IS2,…],[TS1,TS2,…],[FS1,FS2,…]] where ISi,TSi,FSi correspond to species in the initial/transition/final states and the [TS1,TS2,…] list is optional.
- **mode** (*str*) – Output mode. Should be 'latex' for LaTeX, or 'text' for plain text.
- **include_TS** – Include the transition-state in the output. Optional parameter, default is True.

:type include_TS:bool

Parameters **print_out** – Print the reaction to stdout. Optional parameter, default is False.

:type print_out:bool

retrieve_data(*mapp*, *point*, *precision=2*)
Retrieve the data corresponding to a given point in descriptor space from a CatMAP 'map' object. If no data is found for the specified point, then None is returned.

Parameters

- **mapp** (*CatMAP map (see MapperBase)*) – CatMAP "map" structured lists of descriptor points and corresponding values.

- **point** (*[float]*) – Coordinates of a point in descriptor space.
- **precision** (*int*) – Require descriptor coordinates to match with 'precision' decimals. Optional parameter, default is 2.

static reverse_rxn(*rxn*)
Reverse the reaction provided. [[IS],[TS],[FS]] -> [[FS],[TS],[IS]]

Parameters rxn (*[[str]]*) – Reaction in CatMAP form:

- [[IS],[TS],[FS]] for activated reaction
- [[IS],[FS]] for non-activated reaction where IS,TS,FS correspond to the names of the species in the initial/transition/final states respectively.

run(***kwargs*)
Run the microkinetic model. If recalculate is True then data which is re-loaded will be used as an initial guess; otherwise it will be assumed to be correct.

Parameters recalculate (*bool*) – If True solve model again using previous results as initial guess

static same_rxn(*rxn1*, *rxn2*)
Determine if two reactions *rxn1* and *rxn2* are identical.

Parameters

- **rxn1** (*[[str]]*) – Elementary reaction list. See print_rxn for syntax.
- **rxn2** (*[[str]]*) – Elementary reaction list. See print_rxn for syntax.

set_rxn_options()
sets elementary rxn-specific attributes to the appropriate places

single_point_analysis(*pt*)
Find rates/coverages at a single point.

Parameters pt (*[float]*) – Point in descriptor-space ([x,y])

texify(*ads*)
Generate LaTeX representation of an adsorbate.

Parameters ads (*str*) – Adsorbate short-hand.

update(*dictvar*, *override=False*)
Update the attributes of the model with the attribute names/vals included in dictvar. The keys of dictvar correspond to attributes of the ReactionModel to be set, and the values correspond to the values they will be set to.

Parameters

- **dictvar** (*dict*) – Dictionary of key names corresponding to attributes of Reaction-Model instance to be updated with the associated values in dictvar.
- **override** (*bool*) – If True then the values in dictvar will override existing values of the attributes of ReactionModel instance. Optional parameter, default is False.

verify()
Run several consistency check on the model, such as :

- all gas ratios add to 1.
- all mass and site balances are fulfilled.
- prefactors are set in the correct format.

- a mapping resolution is set (the default is 15 data points per descriptor axis).

4.4 catmap.functions module

`catmap.functions.`**`add_dict_in_place`**(*dict1*, *dict2*)

Updates dict1 with elements in dict2 if they do not exist. otherwise, add the value for key in dict2 to the value for that key in dict1

Parameters

- **dict1** (*dict*) – Dictionary.
- **dict2** (*dict*) – Dictionary.

`catmap.functions.`**`cartesian_product`**(**args*, ***kwds*)

Take the Cartesian product

Todo: Explain what the args and kwds are

`catmap.functions.`**`constrained_relaxation`**(*A*, *b*, *x0*, *x_min*, *x_max*, *max_iter=100000*, *tolerance=1e-10*)

Solve Ax=b subject to the constraints that x_i > x_min_i and x_i < x_max_i. Algorithm is from Axelson 1996.

Note that x_min/Max are both lists/arrays of length equal to x

Parameters

- **A** (*numpy.array*) – A matrix.
- **b** (*numpy.array*) – b vector.
- **x0** (*numpy.array*) – x vector
- **x_min** (*array_like*) – Minimum constraints.
- **x_max** (*array_like*) – Maximum constraints.
- **max_iter** (*int, optional*) – Maximum number of iterations.
- **tolerance** (*float, optional*) – Tolerance.

Todo: Check to make sure docstring is correct.

`catmap.functions.`**`convert_formation_energies`**(*energy_dict*, *atomic_references*, *composition_dict*)

Convert dictionary of energies, atomic references and compositions into a dictionary of formation energies

Parameters

- **energy_dict** (*dict*) – Dictionary of energies for all species. Keys should be species names and values should be energies.
- **atomic_references** (*dict*) – Dictionary of atomic reference energies (?)
- **composition_dict** (*dict*) – Dictionary of compositions

Todo: Explain the keys and values for energy_dict, atomic_references, and composition_dict `catmap.functions.`**`fetch_all_output_variables`**`()`

Use code-inspection to extract all processed output variables from

catmap.scalers.scaler_base.ScalerBase.set_output_attrs, catmap.solvers.solver_base.SolverBase.set_output_attrs

New keywords should work out of the box if they are added in one of those functions and using one of the kind of if-statements that are already in place.

`catmap.functions.`**`get_composition`**`(`*species_string*`)`

Convert string of species into a dictionary of species and the number of each species.

Parameters **`species_string`** – A string of the reaction species. Should be a chemical formula string that may also contain '-','&',or,'pe'. 'pe' is a special case corresponding to a proton-electron pair and has the compositon of H, while ele corresponds to an electron and has no associated atoms.

`catmap.functions.`**`linear_regression`**`(`*x*, *y*, *constrain_slope=None*`)`

Perform linear regression on x and y and return the slope and intercept.

Parameters

- **`x`** (*`array_like`*) – x-coordinates.
- **`y`** (*`array_like`*) – y-coordinates.
- **`constrain_slope`** (*`float, optional`*) – Slope constraint

`catmap.functions.`**`match_regex`**`(`*string*, *regex*, *group_names*`)`

Find matching regular expression in string and return a dictionary of the matched expressions.

Parameters

- **`string`** (*`str`*) – String.
- **`regex`** (*`str`*) – Regular expression.
- **`group_names`** (*`list`*) – Corresponding names for each matched group.

Todo: Check that this docstring is correct.

`catmap.functions.`**`numerical_jacobian`**`(`*f*, *x*, *matrix*, *h=1e-10*, *diff_idxs=None*`)`

Calculate the Jacobian matrix of a function at the point x0.

This is the first derivative of a vectorial function:

f : R^m -> R^n with m >= n

Hacked from mpmath.calculus.optimize

Parameters

- **`f`** (*`callable`*) – Function.
- **`x`** –
- **`matrix`** –
- **`h`** (*`float, optional`*) –

Todo: Fill in the descriptions for f, x, matrix, and h `catmap.functions.`**`offset_smooth_piecewise_linear`**`(`*theta_tot*, *slope=1*, *cutoff=0.25*, *smoothing=0.05*, *offset=0.1*`)`

Piecewise linear function with an offset. Not equivalent to piecewise linear for second-order interactions

Parameters

- **`theta_tot`** –
- **`max_coverage`** (*`int`*, *`optional`*) – Maximum coverage.
- **`cutoff`** (*`float`*, *`optional`*) – Cutoff.
- **`smoothing`** (*`smoothing`*, *`optional`*) – Smoothing.
- **`offset`** (*`float`*, *`optional`*) – Offset.

Todo: Fill in description for theta_tot

`catmap.functions.`**`parse_constraint`**`(`*minmaxlist*, *name*`)`

Parse constraints for the relation. Returns two lists of minimum and maximum constraints

Parameters

- **`minmaxlist`** (*`list`*) – List of minimum and maximum constraints.
- **`name`** (*`str`*) – Name for the list of constraints.

Todo: Explain minmaxlist and name

`catmap.functions.`**`scaling_coefficient_matrix`**`(`*parameter_dict*, *descriptor_dict*, *surface_names*, *parameter_names=None*, *coeff_mins=0*, *coeff_maxs=1e+99*, *return_error_dict=False*`)`

Class for determining adsorption and transition-state energies as a linear function of descriptors.

Parameters

- **`parameter_dict`** (*`dict`*) – Dictionary where the key is adsorbate name and the value is a list of adsorption energies for each surface. If some surfaces do not have an adsorption energy use None as a placeholder.
- **`descriptor_dict`** (*`dict`*) – Dictionary where the key is surface name and the value is a list of descriptor values for each surface.
- **`surface_names`** (*`list`*) – List of surfaces which defines the order of surface adsorption energies in parameter_dict.
- **`parameter_names`** (*`list`*, *`optional`*) – List of adsorbates which defines the order of adsorption coefficients in the output. Default is the order of parameter_dict.keys().
- **`coeff_mins`** (*`float`*, *`optional`*) – Defines the minimum value of the coefficient for each descriptor. Should be a matrix/array/list of lists which matches the shape of the expected output.
- **`coeff_maxs`** (*`float`*, *`optional`*) – Same as coeff_mins but for the maximum value of the coefficient.
- **`return_error_dict`** (*`bool`*, *`optional`*) – Specify whether or not to return a dictionary of the errors.

`catmap.functions.`**`smooth_piecewise_linear`**`(`*theta_tot, slope=1, cutoff=0.25, smoothing=0.05*`)`

Smooth piecewise linear function.

Parameters

- **`theta_tot`** –
- **`slope`** (`float, optional`) – slope of line
- **`cutoff`** (`float, optional`) – Cutoff.
- **`smoothing`** (`float, optional`) – Amount of smoothing.

Todo: Fill in descriptions and types for theta_tot

4.5 catmap.cli module

`catmap.cli.`**`get_options`**`(`*args=None, get_parser=False*`)`

`catmap.cli.`**`main`**`(`*args=None*`)`

The CLI main entry point function.

The optional argument args, can be used to directly supply command line argument like

$ catmap <args>

otherwise args will be taken from STDIN.

`catmap.cli.`**`match_keys`**`(`*arg, usage, parser*`)`

Try to match part of a command against the set of commands from usage. Throws an error if not successful.

`catmap.cli.`**`sh`**`(`*banner*`)`

Wrapper around interactive ipython shell that factors out ipython version depencies.

4.6 Module contents

`class` `catmap.`**`ReactionModelWrapper`**

`__getattr__``(`*attr*`)`

Return the value of the reaction model instance if its there. Otherwise return the instances own value (or none if the instance does not have the attribute defined and the attribute is not private)

`__module__ = 'catmap'`

`__setattr__``(`*attr, val*`)`

Set attribute for the instance as well as the reaction_model instance

`catmap.`**`griddata`**`(`**args, **kwargs*`)`

Wrapper function to avoid annoying griddata errors

`catmap.`**`load`**`(`*setup_file*`)`

CHAPTER 5

Troubleshooting

Where to get help when things go wrong :

- post an issue on Github
- mail to the mailling list mkm-developers-request@lists.stanford.edu

In either case by as specific as possible about how to reproduce your problem. Write concrete steps (step 1., step 2., step. 3, ..) required for doing so. Please post details of your installation (operating system, python version, version of used libraries) if you think it may have to do with your problem.

5.1 Frequently Asked Questions

- What does CatMAP stand for ?

  CatMAP is an acronym for **Cat**alysis **M**icrokinetic **A**nalysis **P**ackage. It is also an allusion to Richard Feynman's anecdote regarding a "map of a cat".

- How can I cite CatMAP?

  If you find CatMAP useful in your research please cite:

  A.J. Medford et. al. CatMAP: A Software Package for Descriptor-Based Microkinetic Mapping of Catalytic Trends. Catalysis Letters, 145, 3, pp 794-807 dx.doi.org/10.1007/s10562-015-1495-6

  One goal of CatMAP is to increase the transparency, repeatability, and accessibility of descriptor-based kinetic models. To this end, if you find CatMAP useful in your research we would appreciate it if you make your input files available for others to use. You can do this by including the inputs in the supplementary information of publications, and/or sharing the inputs with developers after publication so that they can be made public.

- How can I contribute to CatMAP?

  We are always happy to have new developers! There is lots of coding to be done on CatMAP for good Python programmers, and lots of documentation for those who don't like programming. If you are interested please get in touch with the developers at mkm-developers@lists.stanford.edu.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index