Workshop Introduction to Kinetic Monte Carlo of Systems (KMCOS)

Mie Andersen, Juan Manuel Lorenzi, Meelod Waheed

January 27, 2022

1 Introduction

1.1 What is Kinetic Monte Carlo

The kinetic Monte Carlo (KMC) method is a simulation technique that allows to achieve a numerical solution to the time evolution of a system that can be described with a Markovian Master equation. That is, the system can exist in a given number of discrete states and can switch between these states according to known transition rates that only depend on the present state of the system. Chemical reactions involve the changing of one state into another. Hence, KMC can be applied as a multiscale modelling technique to study chemical reactions as well as other processes. For example, KMC has been applied to study diffusion in materials, surface catalysis, the growth of crystals, deposition on surfaces, and other problems. General introductions to the KMC method can be found in Refs. [1-4]. Here we will use the open-source KMC code “kmcos”, which is an updated version of the ”kmos” code developed by Max Hofmann [5]. For a recent tutorial introduction to KMC based on the “kmos” / “kmcos” code we refer the reader to Ref. [6].

1.2 Features of kmcos

- User-friendly Python interface
- Viewer GUI (currently working only partially in kmcos)
- Runner API (recommended way of running kmcos)
- Efficient Fortran backends (`local_smart, lat_int, otf`)
- Lateral adsorbate-adsorbate interactions (`lat_int` or `otf` backends)
- Acceleration of quasi-equilibrated processes
- Sensitivity analysis
- Desired timesteps
- Snapshots
- Machine Learning identification of local configurations (under development)

1.3 Installing VirtualBox, Ubuntu, and kmcos

It is recommended to install kmcos on Ubuntu within a Python virtual environment, and our instructions are written accordingly, see Sections 1.4 to 1.7 or:
https://kmcos.readthedocs.io/en/latest/installation/index.html

The kmcos code can be found on: https://github.com/kmcos/kmcos

If you plan to use a Windows machine, it is recommended to first get VirtualBox and to make an Ubuntu Virtual Machine. In that case, please first follow the instructions in the below Sections 1.3.1 to 1.3.6.

1.3.1 Installing VirtualBox

Download VirtualBox with the link provided: https://www.virtualbox.org/wiki/Downloads. Note that VirtualBox 6.1.30 is used and confirmed to work with kmcos.

1.3.2 Setting up Ubuntu on VirtualBox

Open VirtualBox, then click on “New” to create a new virtual machine. Name the Ubuntu machine and select “Linux” as the type and select “Ubuntu (64-bit)” as the version. You will then be prompted to allocate the amount of memory to the Ubuntu virtual machine. We recommend that you allocate at least 8gb of ram if possible. Click next and select “Create a virtual hard disk now,” click create, then select “VHD (Virtual Hard Disk).” Next, we’ll want to dynamically allocate storage on our physical hard disk on the Ubuntu machine. The recommended is 25GB if you do not plan on installing Ubuntu updates later. If you wish to install Ubuntu updates later, we recommend allocating more than 25GB.

Once you create the Virtual Hard Disk, you will be able to see the Ubuntu machine on your VirtualBox dashboard.

1.3.3 Downloading Ubuntu Disk Image File

Download the Ubuntu LTS Desktop file with this link: https://releases.ubuntu.com/20.04/. Note that Ubuntu

20.04 and also 18.04.6 have been used and confirmed to work for kmcos. Later versions of Ubuntu are not confirmed to work. In particular, the Viewer GUI may not work in later Ubuntu versions. Now go back to VirtualBox, click “Settings,” “Storage,” “Empty.” In attributes, click the blue disk icon and add the Ubuntu disk file that was downloaded.

1.3.4 Installing Ubuntu on VirtualBox

Start the Ubuntu machine on VirtualBox and select the iso file you selected previously in settings. Once Ubuntu boots up, select “Install Ubuntu” when prompted on the screen. Select your keyboard layout, then check “Normal Installation” and uncheck “Download updates while installing Ubuntu” when prompted on the next screen. It’s important you uncheck “Download updates while installing Ubuntu” as we cannot guarantee the program will work with the option checked. On the next screen, check “Erase disk and install Ubuntu” and continue. Next, select your current location and set up your name and password on the next screen. After the installation process is completed, restart the Virtual Machine. Log in and you should see the Ubuntu desktop.

As a note: if you want to enter full-screen mode with the virtual machine, type right CTRL + F.

1.3.5 Creating a Shared Drive Between Local and Ubuntu

On the top left corner of the VirtualBox window, click on “Devices,” then “Insert Guest Additions CD Image.”

Within the Ubuntu virtual machine, you should see a “disc” displayed on the left side for the guest additions CD. Open that disc, right click on “autorun.sh” and choose to run this file as a program.

We now want to add a user account to the group vboxsf. Go to the Ubuntu Terminal, and input these commands:

```
sudo usermod -a -G vboxsf "$USER"
```

Note that “$USER” is the name of your user account profile in Ubuntu. Reboot/restart the Ubuntu virtual machine completely.

Now create a shared folder on your Windows side (normally C:\SharedFolderVM)

Click on “Devices,” then “Shared Folder.” Select the folder you want to share between the Ubuntu and Windows PC on the Windows File Manager by clicking on “Other” on the “Folder Path“ dropdown. Go to shared Folder Options, and right-click on “Machine Folders.” Choose “Auto-mount” and “Make Permanent.”

1.3.6 Bidirectional Clipboard Between Local and Ubuntu

Click on “Devices” on the top right, then “Shared Clipboard” and click on “Bidirectional. You can also allow “Shared Drag and Drop” from the same menu as well. You should now have the ability to access the shared folder and bidirectional clipboard between your local and Ubuntu machines.

If the Ubuntu machine provides an error on the Guest Additions or the shared folder feature does not work, input this command to manually install Guest Additions:

```
sudo apt-get install virtualbox-guest-dkms
```

Then reboot the Ubuntu machine again.

If you run into an issue where the Ubuntu machine does not load after resetting, complete multiple full shutdowns and start the Ubuntu machine on VirtualBox.

1.4 Creating a Python Virtual Environment on Ubuntu

For all installations, we highly recommend installing kmcos on Ubuntu within a python virtual environment, and our instructions are written accordingly. Enter these commands in the Ubuntu terminal to create a virtual environment:

```
cd ~
sudo apt-get update
sudo apt-get install python3
sudo apt-get install virtualenv
virtualenv -p /usr/bin/python3 ~/VENV/kmcos
source ~/VENV/kmcos/bin/activate
```

For all installations, be sure to be within the virtual environment created with the command: `source ~/VENV/kmcos/bin/activate`. If you need to exit the virtual environment, enter 'deactivate' in the Ubuntu terminal. It is important that you always remain in the virtual environment when installing the following dependencies and further files in this documentation.

1.5 Installing kmcos on Ubuntu

To fetch the latest version of kmcos using git, enter these commands in the Ubuntu terminal:

```
sudo apt install git
git clone https://github.com/kmcos/kmcos-installers
cd kmcos-installers
bash install-kmcos-linux-venv.bash
```

You can also directly download the repository from https://www.github.com/kmcos/kmcos and then bash install kmcos from the terminal as well.

1.6 Installing Viewer GUI

To start up the viewer GUI, we need to download some dependencies form kmcos-installers. However, note that the Viewer GUI is currently working only partially and may not work at all, depending on the used Ubuntu version (see Section 1.3.3). Type the following (you must change 20 to 18 if using Ubuntu18):

```
cd kmcos-installers
bash install-kmcos-view-linux-venv-Ubuntu20.bash
```

Now we need to cd into …kmcos/examples/MyFirstSnapshot_local_smart and run:

```
ipython
```

Run the following commands in ipython:

```
from kmcos.run import KMC_Model
model = KMC_Model()
model.view()
```

To control the zoom of the atoms, you can add an optional argument: `model.view(scaleA = 6)`. Unfortunately, the atoms are currently not centered within the Viewer Window, which means the you may only see a corner of the surface (depending on the zoom level). We are currently working on fixing this as well as a number of other known issues with the Viewer GUI. While the Viewer can be useful for getting a quick glance of how the surface looks like, the recommended way of running kmcos is through the application programming interface (API), which is introduced in Section 2.

1.7 Frequently Asked Questions

How to Turn off Automatic Updates on Ubuntu

Go to Software & Updates application in 'Show Applications." Click on "Updates" and select these options:

- For other packages, subscribe to: Security updates only
- Automatically check for updates: Never
- When there are security updates: Display immediately
- When there are other updates: Display every two weeks
- Notify me of a new Ubuntu version: Never

How to Create Snapshots on Ubuntu

Open the VirtualBox window and select "Machine -> Tools -> Snapshots." At the snapshots section, you can click "Take" to save the current state of Ubuntu as a snapshot. From here, you have the option to restore your current Ubuntu machine to the snapshot or clone the snapshot to a new Ubuntu machine.

How to Close Windows That Will Not Close, Such as the View Window

Use this command on a new terminal, then click on the window:

```
xkill
```

2 Visualization & Quantitative analysis

2.1 Scripting, Plotting and Visualization

The kmcos API uses the Python programming language. Documentation can be found at: https://docs.python.org/3/. Some basic usage and examples are provided below.

2.1.1 Elements of Python Syntax

1. Printing 'Hello World' on the terminal:

```
print ('Hello World')
```

2. Importing modules:

```
import ... as ...
from ... import ...
```

3. (Some of) python variable types:

Basic Types:

```
some_text = 'This' # strings
some_integer = 134 # integers
some_float = 0.8 # floating point numbers
some_boolean = True # boolean
```

Lists:

```
L = ['a',3 ,4.7]
L[0] # 'a'
L[1] # 3
```

Dictionaries:

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
dict ['Age'] # 7
```

4. If statements:

```
if (condition):
...
elif (condition):
...
else:
...
```

5. Flow control with for loops:

```
for i in range (3):
print(i)
```

Output:

```
0

1
```

2

```
for i, j in enumerate(['hat', 'cat', 'bat']):
print(i + ' ' + j)
```

Output:

```
0 hat

1 cat

2 bat
```

2.1.2 Scientific Computing Using the NumPy Package

The NumPy package contains many useful things such as N-dimensional array objects, linear algebra, and random number capabilities. Documentation can be found at: https://www.numpy.org/. The package is imported and used as follows:

```
import numpy as np
A = np.array([0 , 1])
```

2.1.3 Plotting Using Matplotlib

The Python plotting library matplotlib can be used to create publication quality figures in a variety of formats. Documentation can be found at: https://matplotlib.org/. Any other plotting software can of course also be used, provided the kmcos output data are stored in a convenient format. The advantage of using matplotlib is the easy integration into Python client scripts for running kmcos.

2.1.4 Visualizing atomic structures using ASE

The Atomic Simulation Environment (ASE) is a set of tools and Python modules developed for atomistic simulations. It is used in kmcos to visualize the current state of a model. Documentation can be found at: https://wiki.fysik.dtu.dk/ase/.

2.2 First API Steps: Running kmcos Interactively.

While we can use the GUI viewer to get a general idea of the behavior of the model, quantitative analysis requires the use of the advanced programming interface (API). Be sure that you are inside the kmcos virtual environment before proceeding. You can do so with this command in the terminal: `source ~/VENV/kmcos/bin/activate`.

First, we need to fetch some material for the following tutorials. Within the terminal, cd into your desired directory for saving this material, and enter the following command:

```
git clone https://github.com/kmcos/intro2kmcos
```

Then cd into intro2kmcos/task_material/COoxRuO2_local_smart, which contains a compiled surface catalysis KMC model for the CO oxidation reaction ($2\ CO + O_2 \rightarrow 2\ CO_2$) over the $RuO_2(110)$ surface. You can read more about this model in the original publication [7] and in the tutorial KMC review mentioned above [6]. In brief, the $RuO_2(110)$ surface contains two types of adsorption sites, bridge (br) and cus, as illustrated in Figure 1.

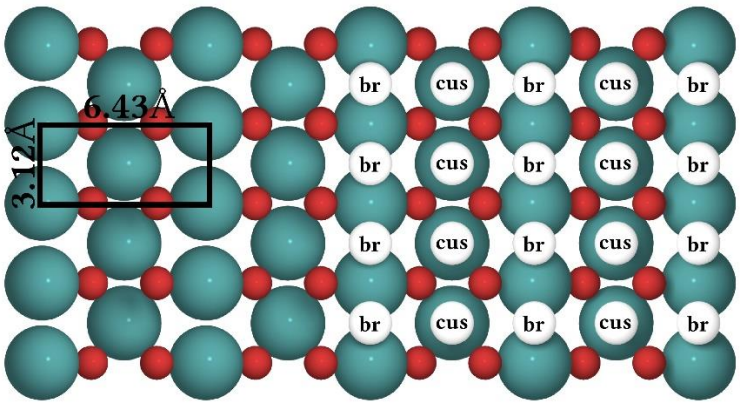

Figure 1. Top view of the structure of the RuO_2(110) surface. To the left, the surface unit cell is shown as a black rectangle. Big green spheres = Ru atoms, small red spheres = O atoms. To the right, the coarse-grained lattice structure is sketched. It consists of alternating columns of cus (coordinately unsaturated) and br (bridge) sites.

The reactant molecules CO and O_2 can adsorb onto both of these site types. CO adsorbs/desorbs into a single site, whereas O_2 adsorbs/desorbs dissociatively into two neighboring sites (i.e. br-br, cus-cus or br-cus). During the simulation, a given site can thus be empty or covered with CO or O. Furthermore, CO and O can diffuse on the surface or react with each other to form CO_2, which is assumed to desorb immediately to the gas phase (since it binds very weakly to the surface).

The Python script that sets up the CO oxidation model is the file: intro2kmcos/task_material/COoxRuO2__build.py. This file uses the ase thermochemistry way of including temperature dependence, as will be explained in section 3.1. In case you would want to modify the model, this is done by modifying the Python script and then saving and compiling the model by running the following command:

```
python3 COoxRuO2__build.py
```

This creates two things: The COoxRuO2.xml file contains the data defining the model in xml format and the directory COoxRuO2_local_smart contains the compiled model. Note that both exist already in the task_material folder and will be overwritten by the above command (only the files defining the KMC model in COoxRuO2_local_smart will be overwritten, not any client scripts, see Section 2.3). In order to launch the API, cd into the COoxRuO2_local_smart directory and run the command:

```
ipython
```

This will initialize `ipython`, an interactive Python interface. Inside this interface, we can load our model by writing:

```
from kmcos.run import KMC_Model
model = KMC_Model()
```

These commands will give some text output (including the model's name and author), and create the model object, which we will use to run our simulations.

A useful shortcut is to skip the above and type: 'python3 kmc_settings.py run' in the folder containing the compiled kMC model, which directly opens the interactive shell and creates the model object.

The most elementary thing we can do is with our model is to directly run it:

```
NSTEPS = 1e6
model.do_steps(NSTEPS)
```

which will run NSTEPS KMC steps (in this case 1 million). This will take some time (probably a few seconds), and then we will recover the (ipython) command prompt.

Below are listed some other useful commands:

1. Change the value of a parameter (e.g. temperature or pressure):

```
model.parameters.T = 550
model.parameters.p_COgas = 0.5
```

2. To print a list of the current values of all parameters and rate constants in the model, simply type:

```
model
```

3. Print current occupations (as averaged coverages):

```
model.print_coverages()
```

4. Print a list of the number of times each process has been executed since the beginning of the simulation:

```
model.print_procstat()
```

5. Analyze the current state of the model:

```
atoms = model.get_atoms()
```

This command returns an ASE atoms object, which can be visualized by typing:

```
model.show()
```

The `atoms` object also contains some additional data:

```
atoms.occupation
atoms.tof_data
atoms.kmc_time
atoms.kmc_step
```

The occupations and turn-over-frequencies (TOFs, i.e. the number of produced molecules, here CO_2, per surface site per second) come in the form of NumPy arrays. In order to get the headers for these arrays, type:

```
model.get_occupation_header()
model.get_tof_header()
```

The TOFs that are affiliated with the atoms object are TOFs averaged over the simulated time since the last `model.get_atoms()` call. In contrast, `atoms.occupation` is the current occupation (identical to what is printed using the `model.print_coverages()` command).

6. Sample an average model and return TOFs and coverages in a standardized format:

```
model.get_std_sampled_data(samples, sample_size, tof_method='integ',
output='str')
```

The parameter `samples` is the number of batches to average over. The number of KMC steps in each batch or sample is given by the `sample_size` parameter. In each case check carefully that the desired observable is sampled good enough! The parameter `tof_method` allows switching between two different methods for evaluating TOFs. The default method 'procstat' evaluates the procstat counter, i.e. simply the number of executed events in the simulated time interval. 'integ' will evaluate the number of times the reaction could be evaluated in the simulated time interval based on the local configurations and the rate constant.

The output of `model.get_std_sampled_data()` is determined by the parameter output, which can be set to 'str' or 'dict'. The default 'str' returns a text string containing first the values of all adjustable parameters, then the TOF(s) and coverages, and finally the total sampled kmc time, the total simulated kmc time (including also the time simulated before the `model.get_std_sampled_data()` call), and the number of sampled KMC steps, all separated by spaces. This text string can be converted to a Python list (L) using the Python `split()` command:

```
data = model.get_std_sampled_data(samples=10, sample_size=1e6)
L = data.split(' ')
print(L)
```

Alternatively, output='dict' returns the above information in the form of a Python dictionary. For example, the TOF for CO oxidation can be retrieved using the command:

```
output_dict = model.get_std_sampled_data(samples=10, sample_size=1e6,
output = 'dict')
print(output_dict)
TOF_CO = output_dict['CO_oxidation']
print(TOF_CO)
```

7. Access to the Fortran modules:

The above commands are often sufficient when running and simulating a kmcos model, but in certain cases direct access to the Fortran data structures and methods is desirable. The Fortran modules base, lattice, and proclist are atttributes of the model instance kmc_model.so. This model instance can be explored using ipython and <TAB>:

```
model.base.<TAB>
model.lattice.<TAB>
model.proclist.<TAB>
```

8. Deallocating memory:

```
model.deallocate()
```

This command is important to call once a simulation has finished, if you want to run a new simulation within the same script, as it frees the memory allocated by the Fortran modules.

Reset model:

```
model.reset()
```

This command is called after `model.deallocate()` in order to reset the model to its initial state.

2.3 Client Scripts

We have already seen that the kmcos GUI is useful to quickly investigate model behavior, and that interactive use of the API allows for finer control and more precise quantitative evaluation. In everyday use however, the most useful way of using kmcos is through client scripts. With client scripts we can automatize the task we performed using the interactive interface.

2.3.1 Generating an Arrhenius Plot

As a first example of a client script, we will see how to build an Arrhenius plot (i.e. log(TOF) vs. 1/T) for the RuO_2 CO oxidation model. This type of analysis is used heavily in the catalysis community for understanding temperature effects on reaction rates and how this relates to the reaction mechanism. In case the reaction mechanism involves a single thermally activated rate-limiting step, the Arrhenius plot will be a straight line, from which the apparent activation energy, E_a , and the pre-exponential factor, A, can be determined, i.e.

$r = A \cdot \exp\left(\frac{-E_\mathrm{a}}{k_\mathrm{B}T}\right)$,

where r is the reaction rate (same as TOF) and k_B is the Boltzmann constant. A detailed discussion of the meaning of the apparent activation energy can be found in Ref. [8].

Be sure that you are in the kmcos virtual environment by entering this command in the terminal: source ~/VENV/kmcos/bin/activate. The example script plot_arrhenius.py can be found in the directory of the compiled model: intro2kmcos/task_material/COoxRuO2_local_smart. You can run it using the following command (note that this may take around 10 minutes to run, depending on your computer):

```
python3 plot_arrhenius.py
```

Note that this script does not save the obtained TOFs. In case you would just want to modify the plot using data obtained previously, it can be useful to save the data to a file. An example how to save and re-load the data is provided in the script plot_arrhenius_w_file.py.

2.3.2 TASK 1: TOF and coverages vs p diagrams

The objective of this task is to write a client script similar to the one from the previous section. In this case, you need to plot both TOF and coverages (discriminated by site type and species) as a function of CO partial pressure, for constant T = 600 K and pO_2 = 0.1 bar (note that the default units for temperature and pressure in kmcos are Kelvin and bar). The plot should cover values within 10^{-1} bar < pCO < 10^2 bar (log-scale should be used). Note that the partial pressures of the gas-phase species affect the model through the rate constants for adsorption, k_{ad}. These are derived from kinetic gas theory and given by the rate of impingement of the gas-phase species i onto the site st of area A_{st},

$$k_{i,st}^{\mathrm{ad}}(T, p_i) = S_{i,st}(T) \cdot \frac{p_i A_{st}}{\sqrt{2\pi m_i k_{\mathrm{B}} T}} ,$$

where p_i and m_i are the partial pressure and mass, respectively, of species i. $S_{i,st}$ is the temperature-dependent sticking coefficient of species i onto the site st, which is often assumed to be unity if no other information is available. You can find all of the rate constant expressions used in the CO oxidation model in the file kmc_settings.py.

You can use the previous example as a basis, and either directly use matplotlib or another plotting tool you know (if available) by writing to a file. The strings that label the output important for this task in the dictionary output from the model.get_std_sampled_data() function are 'CO_oxidation', 'CO_ruo2_bridge', 'CO_ruo2_cus', 'O_ruo2_bridge' and 'O_ruo2_cus'.

The solution to this task can be found in intro2kmcos/solutions/plot_vs_pCO.py.

2.3.3 Relaxing the system

We either begin the KMC simulation with a clean surface or prepare the system in some user-specified initial state (see Section 2.3.5). In any case, this initial system state might be very different from the steady-state system state. It is therefore necessary to run a number of KMC steps to relax the system before any meaningful information about steady-state TOFs and coverages can be obtained. It should thus always be checked that the system has reached steady-state before calling model.get_std_sampled_data()! An example showing how this can be done is provided in the script relaxation.py.

2.3.4 Preparing the initial state of the system

Since it can be difficult to estimate if the system has reached steady state or not, it can be useful to test different initial states of the system to check that the same steady-state solution is always found, independently of the initial state. The state of the system can be modified at any time during the simulation, but most often one would want to prepare a given initial state before beginning the simulation. The occupation of a site <site> can be changed to the species <species> using the command:

```
model.put(site=[x,y,z,model.lattice.<site>], model.proclist.<species>)
```

where x,y,z are the coordinates of the site. The above command can be quite inefficient if changing many sites at once, since each put() call adjusts the book-keeping database. To circumvent this you can use the _put() method instead:

```
model._put(...)
model._put(...)
...
model._adjust_database()
```

Uncomment lines 16-19 in the script relaxation.py for an example of how to use this.

2.3.5 TASK 2: The effect of the initial state

Try relaxing the model from different initial states, e.g. clean, CO@br, O@cus, etc., by doing the corresponding modifications to the script relaxation.py. Also try to vary the number of KMC steps taken in each sample (sample_size) and the number of samples (Nsamples).

2.3.6 TASK 3: Random initial state from guess coverages

In some cases, one might have a good guess of the steady-state system state in terms of (averaged)

coverages. This could for example be obtained by solving the corresponding model in the mean-field approximation (MFA) using rate equations. In case the reader is not familiar with what is implied by the MFA in chemical kinetics, we refer to Ref. [9] for an introduction.

Using a good guess for the final coverages can substantially speed up the time required to relax the system. Since the MFA does not consider lattice inhomogeneity, the best way to convert the MF coverages to a KMC lattice occupation is to assume a random occupation of the lattice sites. In order to do this, the guess coverages must be converted to the number of sites occupied by each species. For each species, the corresponding number of sites is then chosen randomly among the total number of available sites in the system using the Python `random.sample()` method. Take a look at the script relaxation_random_initialization.py and try relaxing the model from different random initial states.

2.3.7 TASK 4: Sensitivity analysis

In a catalytic system, the macroscopic TOF is often controlled by only one or a few microscopic rate constants. This can be quantified using the so-called Degree of Rate Control (DRC) [10]. Several definitions of this concept exist, but here we will be concerned with quantifying how much the change of the rate constant k of a single microscopic process i (+ for forward process, − for reverse process) affects the TOF according to the formula,

$$x_i^+ = \frac{k_i^+}{\mathrm{TOF}} \frac{\partial \mathrm{TOF}}{\partial k_i^+}\bigg|_{k_{j\neq i}^+, K_{j\neq i} k_j^-},$$

where x_i^+ is the DRC of the forward process i, $k_{j\neq i}^+$ are the rate constants of all other forward processes j different from i, $K_{j\neq i}$ are the equilibrium constants (i.e. the ratio between the forward and the reverse process) of processes j different from i, and k_j^- are the rate constants of all reverse processes j (including the reverse process i).

Take a look at the script DRC.py, which calculates the DRC for CO adsorption onto the cus site. The derivative is calculated using finite differences by changing the rate constant by plus and minus 2% (the delta value). Try to play around with the delta value and the number of sample steps used to get an averaged TOF. Hint: finite differences are extremely sensitive to numerical noise and therefore require very long sampling times to sufficiently converge. Does your result match the result for the appropriate process and CO pressure in the lower panel of Fig. 4 in Ref. [11]? Note that there may be smaller differences since the used CO oxidation model is not completely identical to the literature one.

2.3.8 TASK 5: ModelRunner

For some KMC applications you simply require a large number of data points across a set of external parameters (phase diagrams, microkinetic models). For this case there is a convenient class in the file run_ModelRunner.py to work with. For example:

```
from kmcos.run import ModelRunner, PressureParameter,
TemperatureParameter

class ScanKinetics(ModelRunner):
      P_02gas= PressureParameter(1)
      T = TemperatureParameter(600)
      p_COgas = PressureParameter(min=1e-1, max=1e-2, steps = 20)

if __name__ == “__main__”
ScanKinetics().run(init_steps=1e7, sample_steps=1e7, cores=4)
```

This script generates data points over the specified range. Using the PressureParameter or TemperatureParameter assures that the corresponding parameters will be sampled in a log- or reciprocal-scale, respectively. The script above runs several kmcos jobs synchronously (as many as indicated with the cores argument) and generates an output file with results (in this case called ScanKinetics.dat).

TASK: Try to reproduce the Arrhenius plot using the ModelRunner class. Compare the time spent by the original scripts with this one. Use as many cores as your workstation has.

Hint: ModelRunner uses a file to keep track of which calculations took place. If you need to restart the calculations from scratch, you have to remove the file ending in .lock. Also delete the old ScanKinetics.dat file, since otherwise the new calculations will be appended to the old file.

The solution to this task can be found in the files run_ModelRunner.py and plot_ModelRunner.py in the intro2kmcos/solutions directory.

3 Building a kmcos model

Now that we know how to run an existing kmcos model, we will learn how to generate a new model. While doing this we will learn about the way models are abstractly represented. We will also discuss how the models are saved and how the source code necessary for running the simulations is generated and compiled.

3.1 The elements of a kmcos project

A kmcos model is built by putting together several building blocks as illustrated in Figure 2. Those building blocks are found within kmcos.types.

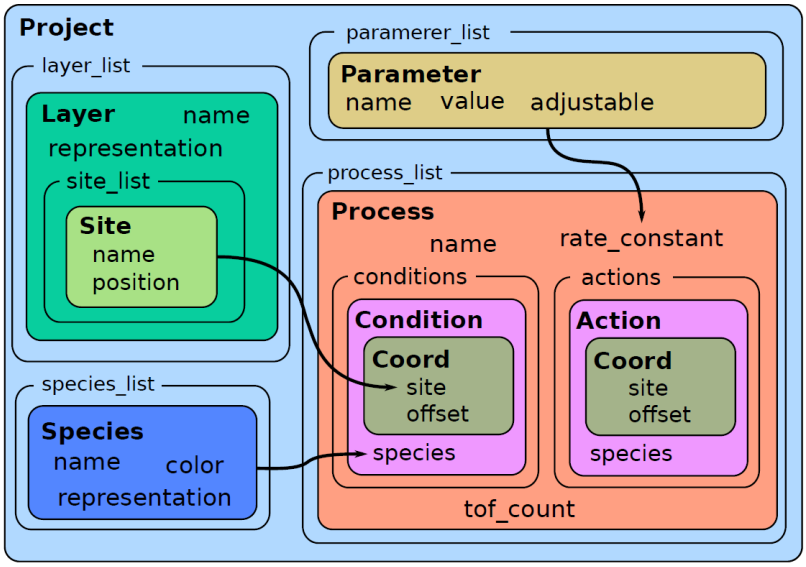

Figure 2. Structure of a kmcos model.

3.1.1 Project

The Project is the structure that contains all other elements.

3.1.2 Meta

Meta contains the meta-data of the project. This includes the model's name, the author's name and email and, importantly, the dimensionality (1D, 2D or 3D) of the system.

3.1.3 Layer

A kmcos Project contains a list of Layer objects. In this tutorial we will only consider models with a single layer, but more could be used. Each Layer has a unique name, an optional representation and a list of sites. Each site has a unique name and a position in the unit cell.

3.1.4 Species

A Project also contains a list of Species. Each species needs to have a unique name. A color and a representation can additionally be added; these are useful while using the Editor and the GUI Viewer, respectively.

3.1.5 Parameter

A list of parameters is included in a Project. Each parameter needs a unique name and a default value. Parameters are used in the definition of the rate constants, and their values can be modified at run-time using the API or (if defined as adjustable) the Viewer GUI.

3.1.6 Process

The processes are perhaps the most complex structure in a kmcos model, and they are built using the elements previously discussed. Apart from a unique name, processes are composed of a list of conditions, a list of actions and a rate constant expression. Optionally, a process can also contain a tof_count attribute if it contributes to some of the turnover frequencies.

Condition: A condition is defined by selecting a single species and a single coordinate. A coordinate is a site, but containing additionally information about its offset, i.e. the relative position of this coordinate with respect to a reference central coordinate. The list of conditions of a process encodes the necessary (local) occupation pattern necessary for the process to occur.

Action: Like conditions, actions are also composed of a species and a coordinate. The list of actions of a process determines what is changed in the system state if the process is actually executed.

Rate constant: Each process needs to be assigned a rate constant expression. These are given as strings that can contain common mathematical expression, any of the user-defined parameters as well as some other predefined parameters and constants. All rate constants are evaluated when the model is first loaded and whenever the value of a parameter is changed.

3.1.7 Thermochemistry (optional)

The enthalpy and entropy of a species (and thus of a reaction, and thus of activation energies) change as a function of temperature. These temperature dependent effects on the activation energies can thus be captured using thermochemistry terms inside of the activation energies of the rate expressions. Kmcos currently has two types of thermochemistry methods, we provide examples of both in this repository. There is a janaf way and an ase way (for the ase way link, the correct place to look is the "formation energy", which provides some explanation in addition to the explanation below). These wordings of "janaf way" and "ase way" are informal wordings for simplicity.

The janaf approach is roughly as follows: there are empirical equations (in particular, Shomate equations) for describing the enthalpy and entropy temperature dependences of gas molecules using multi-term polynomial expressions – these take into account deviations from ideality. These are typically from experimental measurements and are very accurate. They can be extended to surface species. The janaf tables data files are essentially tables of coefficients that are equivalent to the Shomate equation (there is also a similar approach for NASA polynomials, which is not supported by kmcos, though it is possible to convert NASA polynomials to the janaf format). Example file: COoxRuO2__build_w_janaf.py

The ase approach is roughly as follows: there are statistical mechanics equations which can be used to describe the most strongly contributing terms to the temperature dependence, which are entropy terms. These entropy terms are related to vibrations, rotations, and translations. The ase way currently uses some crude approximations for the entropy terms, with parameters that are calculated from electronic structure calculations. Despite its crudeness, the ase way is sufficiently accurate for most studies (as will be further explained in Task 14). Example file: COoxRuO2__build_w_ase.py

3.2 A model step-by-step: O2 adsorption / desorption

We will learn about the components of a kmcos models through a simple example: A model for oxygen adsorption and desorption onto a fcc(100) (square) lattice.

3.3 TASK 7: The ZGB Model

The ZGB model [12] is a classic example of a kinetic Monte Carlo model of CO oxidation. It includes dissociative oxygen adsorption, molecular CO adsorption and a CO-O reaction to form gaseous CO2. The only parameter in the model is the fraction of CO in the gas mixture yCO. The CO adsorption rate per unit cell on empty sites is equal to yCO. The Oxygen adsorption rate per unit cell is (1 − yCO). Desorption of the reactants (CO and O2) is neglected. To avoid deadlocks,

i.e. states in which no process can happen, desorption process with very small rate (i.e. 10−10s−1 should be included. CO2 formation has "infinite" rate constant in the model, which can be modeled by using a very large value (i.e. 1010s−1).

Hint: Beware of double counting when implementing the oxygen adsorption process.

3.4 Modeling (lattice) diffusion

3.4.1 A simple ion diffusion model

Another type of system that can be modeled with kmcos is that of a set of particles diffusing on a lattice. Here we will consider a simple example of particles diffusing on a 2D square lattice. The example render script render_LGD.py can be found in the task_material folder.

A similar simple model in a realistic context (Li diffusion in graphene) can be found in Ref. [13].

3.4.2 Preparing the system

If used as given, trying to run the model will not work. This is because at start-up the system willbe completely empty, a state in which no process can occur. To solve this problem there are two possibilities, depending on our objectives:

- One option is to prepare the system so that it contains a certain number of ions at the start, as was done in section 2.3.4. This is useful if one ones to study the relaxation of the system from the

selected configuration, or to study equilibrium properties.

- Alternative, it is possible to impose special boundary conditions to the system. In this way it is also possible to study steady-state behavior under non-equilibrium conditions.

In this course, we will focus only on the second case.

3.4.3 TASK 8: Implement the boundary conditions

The objective of this task is to set up the boundary conditions for the diffusion model. This can be achieved by including a pair of auxiliary species (’source’ and ’drain’) and the corresponding entry and exit processes (as shown in the slides). Additionally, the kmc_settings.py file needs to be modified in two ways:

- First, the default system size should be changed from the default (20 × 20) to something that is longer in the direction of prominent diffusion, e.g. (50 × 20)
- Second, the setup_model function should be modified to place the sources and drains in the correct positions.

If you manage to set this up correctly, the model can be visualized using model.view() in ipython. To control the zoom of the atoms, you can add an optional argument like this: model.view(scaleA = 6). Be sure that you are in the kmcos virtual environment by entering this line on the terminal: `source ~/VENV/kmcos/bin/activate`.

3.4.4 TASK 9: Extending and testing the lattice diffusion model

Extend the diffusion model to include the effects of an external electric field. For this, introduce a new parameter that measures the strength of the field, and modify the diffusion processes accordingly. Once this is set in place, test how the effects of field strength and particle concentration in the current (details in the slides).

3.5 Lateral interactions in kmcos

Up to now we have only considered models with a relatively small number of processes. In some situations, however, it is possible that the rate constants depend not only on the class of process being executed, but also on the local environment around the adsorbates. For example, the particles in our diffusion problem could interact repulsively, changing the rates of diffusion. In these cases we say that there are lateral interactions in the system.

The standard way to treat this in kmcos is to explicitly incorporate all different processes arising from the interactions. In order to do this, we can use the itertools python module, to programmatically explore possible local states. An example of this is presented in the slides.

3.6 TASK 10: Solid-on-solid crystal growth model

Another problem that can be studied with kinetic Monte Carlo is that of crystal growth. We will consider a very simple model of crystal growth: the Solid-on-Solid (SOS) model. The model we will consider only includes adsorption and desorption processes, neglecting diffusion. The adsorption rate is constant, but the desorption rate is affected by both the system temperature and by lateral interactions (details in the slides).

Interesting examples of use of kinetic Monte Carlo in studies of growth processes can be found in Refs. [14-16].

TASK: Implement the SOS model in kmcos. Simulate crystal growth for two different temperatures (350 K and 450 K) and observe the resulting structures in both cases.

3.7 TASK 11: Diffusion in the SOS model

Implement diffusion processes in the SOS growth model. Compare growth patterns in the model with and without diffusion. Consider also a model without desorption (only adsorption and diffusion).

3.8 TASK 12: Lateral interactions in the diffusion model

Recalculate the plots obtained in TASK 9 for the diffusion model that includes lateral interaction. Do this for different values of the lateral interaction strength.

3.9 TASK 13: Defects in the diffusion model

Extend the 2D diffusion model by adding a 'defect' species, that blocks diffusion. Generate an initialization script that prepares the system with defects randomly located. Test the effect of the presence of defects in the current. Do this for different defect concentrations.

3.10 TASK 14: Compare the energy differences for the ase way versus the janaf way for one system

The way that we will compare the ase way vs. the janaf way is to calculate the rate constants for desorption of CO with each method as a function of temperature and see how large of a difference in energy these rates would correspond to. We will do the same for O2. We will use the files COoxRuO2__build_w_ase.py and COoxRuO2__build_w_janaf.py.

We will check the temperatures: TemperaturesToCheck = [100, 300, 500, 800]
We can change the temperature using the syntax: model.parameters.T = T_value
Then check the rate constant at each temperature using syntax like:
model.rate_constants.by_name('CO_desorption_bridge')

Inside the solutions directory, we have:
COoxRuO2_w_janaf_local_smart
COoxRuO2_w_ase_local_smart
Inside each of those directories is a runfile that produces StackedArray.csv
The two directories StackedArray.csv values are consolidated in ThermoChemistryComparison.xlsx of the solutions. Inside this excel file, the ratio of the ASE rate divided by the JANAF rate is taken. The ASE rates are lower than the JANAF rates, indicating that the ASE rates have a higher barrier (so more high energy gas states). For each of the temperatures investigated, we then compare this ratio the expression e^(-dE/RT) where dE is difference in energies. Looking at values of 1000 J/mol to 15,000 J/mol in the columns to the right hand side of the excel file, we see that the ASE way differs from the JANAF way by around 10,000 J/mol (+/- 5000 J/mol) for this system with the current settings. While this is significant, it is considered within the uncertainties of DFT calculations which are around 20,000 J/mol for this application. Thus, we can say that the ASE way has been implemented sufficiently correctly here, and is adequately accurate for research purposes.

There are some caveats about the entropy approximations of the ASE way. In addition to the entropy approximations, there is also the Zero Point Energy (ZPE) correction. The JANAF way has Zero Point Energy correction inherently within it, for the gas phase values. The ASE way does not. In principle, KMCOS may later add the Zero Point Energy correction but currently does not. The ZPE correction is not so important for small molecules (<5 atoms) but can become more important for larger molecules. The enthalpy corrections (ZPE as well as the population of higher-energy translational, vibrational and rotational energy states at finite temperatures) can actually also be quite large for high-temperature catalysis. See e.g. Fig. 3 in this paper on graphene synthesis at Cu, which were calculated using more sophisticated ASE calculations: http://pubs.acs.org/doi/10.1021/acs.jpcc.9b05642 Additionally, for the ase way, KMCOS currently assumes all vibrational modes are harmonic and ignores rotational modes. Furthermore, in this example, the vibrational modes of the adsorbate were neglected (which brings the adsorbate energy a little closer to the gas phase energy). It is better to include the adsorbate vibrational modes also. This is possible in kmcos by adding species into the species class at kmcos\kmcos\species.py before simulation or during runtime. In the future, such species information will be addable during model building and will then become part of the xml and kmc_settings so that kmcos can add the species to the Species class during runtime.

4 References

Made with endnote:

1. Chatterjee, A. and D. Vlachos, *An overview of spatial microscopic and accelerated kinetic Monte Carlo methods.* Journal of Computer-Aided Materials Design, 2007. **14**(2): p. 253-308.

2. Michail, S., *Kinetic modelling of heterogeneous catalytic systems.* Journal of Physics: Condensed Matter, 2015. **27**(1): p. 013001.

3. Reuter, K., First-Principles Kinetic Monte Carlo Simulations for Heterogeneous Catalysis:

Concepts, Status and Frontiers, in Modelling and Simulation of Heterogeneous Catalytic Reactions: From the Molecular Process to the Technical System, O. Deutschmann, Editor. 2011, Wiley-VCH: Weinheim. p. 71-112.

4. Voter, A., *Introduction to the kinetic Monte Carlo method*, in *Radiation Effects in Solids*, K. Sickafus, E. Kotomin, and B. Uberuaga, Editors. 2007, Springer Netherlands. p. 1-23.

5. Hoffmann, M.J., S. Matera, and K. Reuter, *kmos: A lattice kinetic Monte Carlo framework.* Comput. Phys. Commun., 2014. **185**(7): p. 2138-2150.

6. Andersen, M., C. Panosetti, and K. Reuter, *A Practical Guide to Surface Kinetic Monte Carlo Simulations.* Front. Chem., 2019. **7**(202).

7. Reuter, K., D. Frenkel, and M. Scheffler, *The Steady State of Heterogeneous Catalysis, Studied by First-Principles Statistical Mechanics.* Physical Review Letters, 2004. **93**(11): p. 116105.

8. Mao, Z. and C.T. Campbell, Apparent Activation Energies in Complex Reaction Mechanisms: A Simple Relationship via Degrees of Rate Control. ACS Catalysis, 2019. **9**(10): p. 9465-9473.

9. Temel, B., H. Meskine, K. Reuter, M. Scheffler, and H. Metiu, *Does phenomenological kinetics provide an adequate description of heterogeneous catalytic reactions?* The Journal of Chemical Physics, 2007. **126**(20): p. 204711.

10. Stegelmann, C., A. Andreasen, and C.T. Campbell, *Degree of Rate Control: How Much the Energies of Intermediates and Transition States Control Rates.* J. Am. Chem. Soc., 2009. **131**(23): p. 8077-8082.

11. Meskine, H., S. Matera, M. Scheffler, K. Reuter, and H. Metiu, *Examination of the concept of degree of rate control by first-principles kinetic Monte Carlo simulations.* Surface Science, 2009. **603**(10–12): p. 1724-1730.

12. Ziff, R.M., E. Gulari, and Y. Barshad, *Kinetic Phase Transitions in an Irreversible Surface-Reaction Model.* Physical Review Letters, 1986. **56**(24): p. 2553-2556.

13. Lehnert, W., W. Schmickler, and A. Bannerjee, The diffusion of lithium through graphite: a Monte Carlo simulation based on electronic structure calculations. Chemical Physics, 1992. **163**(3): p. 331-337.

14. Kratzer, P., E. Penev, and M. Scheffler, *First-principles studies of kinetics in epitaxial growth of III–V semiconductors.* Applied Physics A, 2002. **75**(1): p. 79-88.

15. Kratzer, P. and M. Scheffler, Reaction-Limited Island Nucleation in Molecular Beam Epitaxy of Compound Semiconductors. Physical Review Letters, 2002. **88**(3): p. 036102.

16. Šmilauer, P., M.R. Wilby, and D.D. Vvedensky, *Reentrant layer-by-layer growth: A numerical study.* Physical Review B, 1993. **47**(7): p. 4119-4122.