MonteCoffee: A programmable kinetic Monte Carlo framework

MonteCoffee: A programmable kinetic Monte Carlo framework

Mikkel Jørgensen[a] and Henrik Grönbeck[b]
Department of Physics and Competence Centre for Catalysis, Chalmers University of Technology, 412 96 Göteborg, Sweden

(Received 29 June 2018; accepted 31 August 2018; published online 18 September 2018)

Kinetic Monte Carlo (kMC) is an essential tool in heterogeneous catalysis enabling the understanding of dominant reaction mechanisms and kinetic bottlenecks. Here we present MonteCoffee, which is a general-purpose object-oriented and programmable kMC application written in python. We outline the implementation and provide examples on how to perform simulations of reactions on surfaces and nanoparticles and how to simulate sorption isotherms in zeolites. By permitting flexible and fast code development, MonteCoffee is a valuable alternative to previous kMC implementations. *Published by AIP Publishing.* https://doi.org/10.1063/1.5046635

I. INTRODUCTION

Understanding catalytic reaction kinetics is an essential component to develop the field of heterogeneous catalysis. Reaction kinetics can be evaluated through kinetic measurements yielding a macroscopic view of the reaction, or through computer-aided simulations that serve to understand microscopic reaction mechanisms. In particular, methods based on electronic structure calculations have become increasingly important over the past decades.[1–3] A general approach is to link electronic structure calculations with mean-field kinetics, which allows for understanding the average kinetic behavior of uniform catalysts, where the kinetics can be formulated in terms of coverages. However, for systems with different types of sites, such as nanoparticles, there is a need to follow the reaction in greater detail. A direct way of achieving this is by solving Newton's equations of motion, as is achieved by Molecular Dynamic (MD) simulations. However, MD simulations are of limited use in reaction kinetics as the time steps are on the order of vibrational frequencies, which renders chemical reactions rare events. For example, in CO oxidation to CO_2, an adsorbed CO molecule makes about 10^{11} vibrations per CO_2 formation step. To overcome this time scale separation issue, kinetic Monte Carlo[4,5] (kMC) is often used to simulate reaction kinetics. In kMC, a number of reactive events are defined, and their occurrences are simulated by random number generation. Thus, kMC simulations coarse-grain phase-space to exclude vibrations and solely focus on chemical transitions.

In the field of heterogeneous catalysis, several kMC implementations use the lattice kMC method,[6–11] which represents the system by a lattice of sites. One challenge in lattice kMC is to model dynamic site-changes, such as redox reactions, which has been tackled using multi-lattice approaches.[2,6] Moreover, while a lattice-based approach provides a simple representation of the sites, the method is cumbersome to apply to complex geometries such as nanoparticles. As an alternative to lattice kMC, graph-theoretical codes have been developed,[11,12] where a global neighbor list is used to represent the site connectivity. This permits simulations of complex geometries. In the present implementation, the focus is on code flexibility, which is desirable as each system where kMC is applied often is unique. We present a simple and fully flexible framework that allows for rapid kMC code development.

MonteCoffee is an open-source object-oriented programmable (OOP) application,[13] which has the advantage of quick code development and extensibility. The code is written in python as this is a popular high-level programming language, which is malleable and easy to comprehend. Additionally, python has the advantage that it can be combined with other programmable applications such as the Atomistic Simulation Environment[14] (ASE). With an object-oriented approach, a number of challenging tasks are straightforward, such as advanced evaluation of adsorbate-adsorbate interactions, dynamic system variations, and descriptor-based energy landscapes. The code uses neighbor-lists to represent the site connectivity, which is more flexible than the lattice kMC method. Using neighbor-lists is similar to the graph-theoretical approaches; however, in MonteCoffee, the user directly controls the site-connectivity. The code implements the first-reaction method (FRM) algorithm[15] and other algorithms can straightforwardly be implemented.

In this paper, the basic structure and functionality of the MonteCoffee framework is outlined. The main focus is the code implementation and possibilities for extension. The capability of the code is demonstrated in two different examples, namely, CO oxidation over a nanoparticle with a descriptor-based energy landscape and adsorption isotherms in a zeolite.

II. KINETIC MONTE CARLO

kMC simulations typically discretize phase-space into a set of energy basins, which are separated by chemical transitions. This avoids simulating vibrations and focuses on

[a]Electronic mail: mikjorge@chalmers.se
[b]Electronic mail: ghj@chalmers.se

the chemical reactions.[15] Thus, in kMC simulations, a system is defined as a set of sites where a number of reactive events can proceed. Event execution is simulated in time, based on random number generation. This achieves the basic purpose of kMC simulations, to solve the chemical master equation

$$\frac{dP_\alpha}{dt} = \sum_\beta W_{\alpha\beta}P_\beta - W_{\beta\alpha}P_\alpha, \quad (1)$$

where P_α is the probability for the system being in state α and $W_{\alpha\beta}$ is the transition rate from state β to α. Here, a state is defined by a specific occupation of the sites. From the generated states and transitions, data such as turnover frequencies (TOFs) and coverages can be extracted and the detailed reaction trajectory can be followed.

The MonteCoffee software implements the First Reaction Method[15] (FRM). FRM is similar to the Variable Step Size Method (VSSM), also called the n-fold way.[15] FRM executes events chronologically based on randomly generated times of occurrence,

$$t_{\alpha\beta} = t - \frac{\ln u}{W_{\alpha\beta}}, \quad (2)$$

where $t_{\alpha\beta}$ is the time the event occurs, t is the current time, and u is a random uniform number in $[0, 1[$.

The FRM algorithm consists of three main parts, which are illustrated in Fig. 1:

1. **Initialize simulation (T,p), generate reaction times, set time = 0**: This part sets the reaction conditions and initializes the system, sites, coverages, and lists used for bookkeeping. The lists for bookkeeping can include coverages, time steps, the sites where events occurs, etc. Moreover, the initial times of event occurrences are generated, and the simulation time is set to 0.
2. **Perform next event in queue**: Performs the chronologically next event according to Eq. (2).
3. **Advance time. Update local event-list**: The simulation time is updated to the occurrence time of the performed event. Performing the event modifies the lattice locally and nearby events are affected. To enhance performance, the event-list is only updated locally. If nearest-neighbor adsorbate-adsorbate interactions are used, the sites are updated two nearest neighbor distances away from the site where the event was performed.

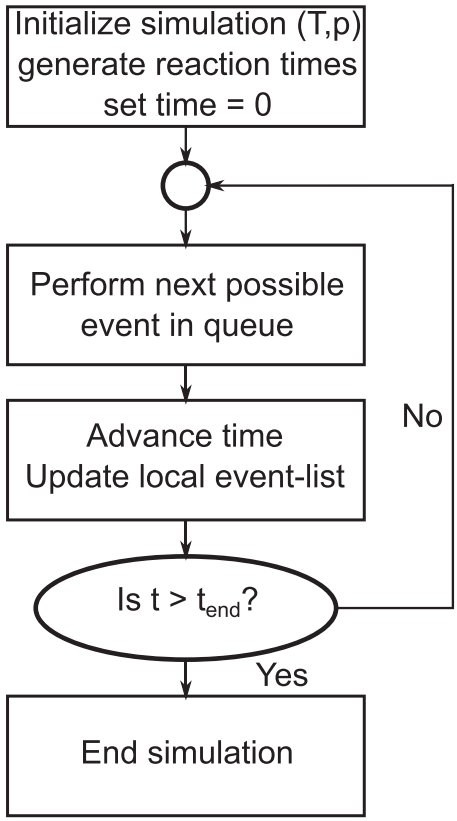

FIG. 1. The first reaction method algorithm.

The loop continues until the simulation has reached its end-time t_{end}. During a simulation, the lists that track statistics grow considerably. Thus, memory is freed at regular intervals by writing these lists to disc as python pickle-files.

kMC simulations are often challenging due to major differences in reaction rates, where slow events are only observed after accelerating the simulation. A rigorous acceleration scheme slowing down individual quasi-equilibrated elementary steps was presented by Chatterjee and Voter[16] and a proof justifying this method was given in Ref. 17. The task of accelerating simulations is relatively simply achieved in MonteCoffee as it uses custom functions to get rate-constants and perform the events. Two acceleration schemes are available in the code: Slowing down fast events by raising energy barriers manually and the generalized temporal acceleration scheme of Dybeck *et al.*[18] The generalized temporal acceleration scheme involves artificially slowing down quasi-equilibrated reaction channels to increase the time step of the simulations.

III. OBJECT-ORIENTED PROGRAMMING

The current implementation uses object-oriented programming (OOP), which is programming centered around objects and classes. A class is a data-structure that contains variables and functions and objects are specific instances of a class. For example, one can define a `Particle` class and instantiate a specific particle object by calling `octahedron = Particle(symbol='Pt',a0=4.00)`. In this case, an octahedral Pt nanoparticle object is instantiated with a lattice constant of 4 Å. After instantiating an object, its variables can be altered during code execution, e.g., as `octahedron.symbol = 'Au'`, which changes the element from Pt to Au. Classes can also contain functions that are called to change the object. For example, a function `set_a0(a_new)` can be defined for the specific particle to change the lattice constant to 3.98 as `octahedron.set_a0(3.98)`.

In MonteCoffee, class inheritance is used, which means that a base-class passes on some variables and functions to derived classes. In the present implementation, inheritance is used in a manner where base classes are templates for the derived classes. This gives a good overview of the tasks involved in defining a simulation. For example, all events defined by the user are derived from a template-class named `EventBase`, which ensures that all user-defined events has

a function that calculates the rate-constants. Thus, if the user-defined derived event class does not contain such a function, the code will give an error message.

It is out of scope to discuss OOP in further detail as it is an extensive subject, with different implementations depending on the programming language. The interested reader is referred to the python documentation[19] for further information about the use of OOP in python. In Secs. IV–VI, classes are named beginning with an uppercase letter (e.g., Site), whereas objects start with a lowercase letter (e.g., site).

IV. CLASSES AND FUNCTIONS

Figure 2 shows a general outline of the code modules. The Monte Carlo engine is in the NeighborKMC module, which communicates with a set of different modules, and handles settings as well as simulation logging. The NeighborKMC module relies on a list of events and a system that is composed of a set of sites.

The MonteCoffee software provides classes and functions to define and structure kMC simulations. To adapt the code to a specific problem, classes are defined by the user using inheritance. Figure 3 shows the main classes, essential variables, and functions in the code. This list of variables is restricted to those that are relevant to the present discussion.

A. NeighborKMC class

The NeighborKMC class performs the simulations and is responsible for the main functionality of the code. It contains a number of variables: `parameters` is a python-dictionary containing keys and values for temperatures, pressures, and other parameters, which are passed on to the events. All key-value pairs in `parameters` are written in the beginning of the simulation log and it can conveniently be used to name or comment a simulation. `options` is a list of settings loaded from a separate file, which contains the frequency of data saving, filenames, etc. `time` and `tend` are floats describing the current simulation time and the simulation end-time, respectively. A central object for the NeighborKMC class is the `system`, which is the `System` instance that the simulation is modeling. `events` is the list of reactive events, and `frm_times` is the list of generated times of occurrence for the events. `events` is populated with `Event` objects in the order they are defined. `basin_vars` represent a set of variables used to steer the generalized temporal acceleration scheme.[18]

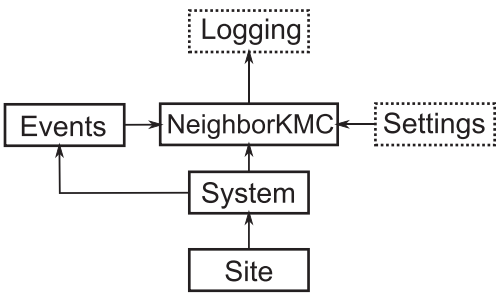

FIG. 2. The main modules and their relationships.

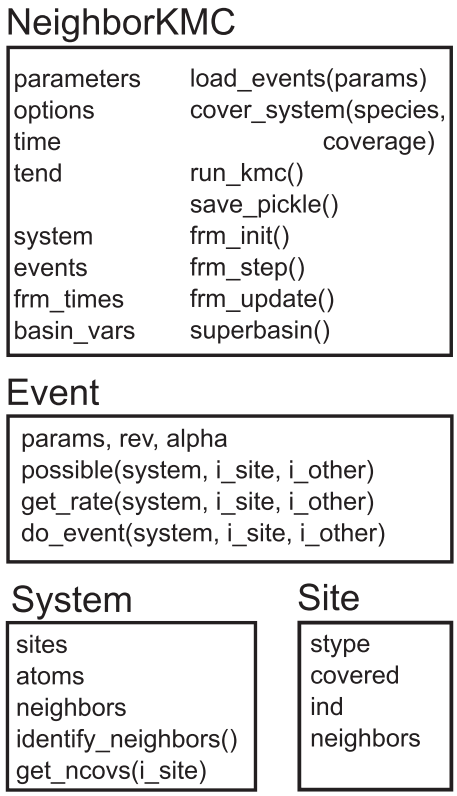

FIG. 3. The main classes, their variables, and functions.

The functions of the NeighborKMC class are mainly used to perform the simulation. Upon instantiating a NeighborKMC object, `load_events(params)` is called to populate the list `events` with the specified parameters such as temperature and pressures. `params` is a subset of `parameters` as discussed below. When instantiating a NeighborKMC object, `cover_system(species,coverage)` is also called to provide a certain coverage of surface species. `run_kmc()` sets time equal to zero and runs the kMC algorithm until `time` has reached `tend`. The function `frm_init()` initializes the kMC algorithm by generating the initial times of occurrences for the events. `frm_step()` takes a Monte Carlo step by performing the next possible event in the queue. After each step, `frm_update()` is called, which searches for events that have been enabled or disabled as a consequence of the performed step. `superbasin()` is responsible for handling the generalized temporal acceleration scheme by slowing down the quasi-equilibrated events during a simulation.

B. Event class

A kMC simulation proceeds using random numbers to perform different reactions and here these events are defined using classes. An `Event` class has three variables. A list of user-defined parameters (`params`) used to calculate the rate-constants. Examples of relevant parameters are pressures, temperatures, and molecular masses. `params` differs from the `parameters` variable of the `NeighborKMC` class as it only contains the relevant parameters for the `Event` class. The second variable is `rev`, which is an integer describing the index of the reverse reaction. The third variable is `alpha`,

which is the multiplication factor used to slow down quasi-equilibrated events in the generalized temporal acceleration scheme.

The class contains three functions: One is a function `possible(system, i_site, i_other)` that during a simulation determines if the event can occur at site number `i_site` and co-participating sites indices `i_other`. The next function `get_rate(system, i_site, i_other)` is a user-defined function that, during simulation, returns the rate-constant for the given event on the specified sites. Finally, `do_event(system, i_site, i_other)` performs the event by modifying the relevant `site` objects in the system.

The user defines the types of reactive events by defining the event-types from the template base-class (`EventBase`) using inheritance. An example is to define `COAdsEvent(EventBase)` as a class that handles CO adsorption events. The parenthesis `(EventBase)` means that `COAdsEvent` is a class derived from `EventBase`. In this example, the `possible(system, i_site, i_other)` function returns `True` if the site is empty, and the `get_rate(system, i_site, i_other)` returns a rate-constant from collision theory. `do_event(system, i_site, i_other)` changes the site occupation of `i_site` to be occupied by a CO molecule.

Reaction energies used to calculate the rate-constants are stored in python dictionaries, which are connected to each user-defined event class. It is left up to the user to define how the rate-constants and energies are calculated and also to ensure thermodynamic consistency. The energies can be functions of occupations and alternative custom properties such as coordination numbers or site-types. The entropy changes during reactions and partition functions are also conveniently stored as member variables of the event-classes.

C. System and site classes

The `System` class contains three variables: A list of `Site` objects (`sites`), an optional `ase.Atoms` object (`atoms`), and a neighbor list to keep track of site-connectivity (`neighbors`). The `sites` list is used during the simulation to manipulate site-properties, such as coverage. `atoms` is used to associate ASE atoms with the kMC simulation, which can be used for visualization of the kMC run. The function `identify_neighbors()` identifies neighboring-sites that are connected and assigns them to the `Site` objects. This function is user-defined and it is typically based on distances between atoms in the `atoms` object. `neighbors` is the global neighbor-list consisting of the individual neighbor-lists of the sites. Thus, `neighbors` defines the global connectivity pattern. `get_ncovs(i_site)` returns the coverage of the nearest neighbors to the site with index `i_site`.

The `Site` class defines a site by the following standard variables: `stype` is a user-defined integer describing the type of site, which is used to distinguish the types of sites, such as edges and corners. If desired, additional variables can be added to describe the reaction energy landscape. For example, each site can be given a coordination number as a reaction energy descriptor and a strain to perturb the reaction energies. `covered` is an integer variable that describes the occupation of the site. The meaning of this integer is decided by the user, where, for example, `covered = 0` can represent empty sites and `covered = 1` a CO-covered site. Multidentate species can be implemented by changing the covered integer to a list `covered = [other_site_index, species_integer]`. `ind` is a list of integers used to associate the site with an `ase.Atoms` object for visualization purposes. `neighbors` is a list of integers that contains the site index of neighbors-sites connected to the site. The indices in `neighbors` are not related to `ind`, but are instead associated with the global site list of the `system` object.

D. Simulation setup

To set up a simulation, the user needs to perform two initial steps. Step 1: Define the list of sites that defines the `System`. When defining the system, the function `identify_neighbors()` must be implemented. For example, this is achieved using the nearest neighbor distance between surface atoms in an `ase.Atoms` object. Step 2: Define one derived event-class for each possible elementary step in the reaction scheme. The specific event-objects are instantiated automatically at run time, and only the classes must be defined by the user prior to the simulation. In this step, reaction energies, entropic barriers, and rate-constants are defined.

V. EXAMPLES

The program has been used in different applications,[20–22] and the following two examples are included to describe the workflow to set up a kMC simulation with MonteCoffee. Reaction energies in these examples are obtained by Density Functional Theory[23,24] (DFT) calculations, with details presented in the Appendix.

A. CO-oxidation on nanoparticles

CO-oxidation over Pt nanoparticles is modeled by the following reaction scheme:

$$\begin{aligned} \mathrm{CO(g)} + * &\leftrightarrow \mathrm{CO}^*, \\ \mathrm{CO}^* + * &\leftrightarrow * + \mathrm{CO}^* \quad \text{(Diffusion)}, \\ \mathrm{O_2(g)} + 2* &\leftrightarrow 2\mathrm{O}^*, \\ \mathrm{O}^* + * &\leftrightarrow * + \mathrm{O}^* \quad \text{(Diffusion)}, \\ \mathrm{CO}^* + \mathrm{O}^* &\rightarrow \mathrm{CO_2(g)} + 2*, \end{aligned} \tag{3}$$

where $*$ denotes an empty site. In this example, the sites are conveniently defined using an `ase.Atom` object. First an octahedral cluster is instantiated as an `ase.Atom` object by calling the following:

```
from ase.cluster import Octahedron
atoms = Octahedron('Pt',
                   length=18,
                   cutoff=6,
                   latticeconstant=4.00).
```

This instantiates an octahedral Pt nanoparticle with a lattice constant of 4 Å. The atoms object can be understood as a list

containing `ase.Atom` objects with information about chemical symbols, positions, charges, etc. Further details can be found in the ASE documentation.[25]

The generalized coordination number[26,27] ($\overline{\text{CN}}$) of the sites is used to distinguish the types of sites. The $\overline{\text{CN}}$ of a site is an extension of the conventional coordination number that is defined by the first nearest neighbor coordination. One site is defined for each surface atom on the particle, which is identified by having a conventional coordination number <12. The site in this example is a coarse-grained entity that entails ontop, bridge, and hollow positions. Figure 4 shows the $\overline{\text{CN}}$ of ontop sites on the considered truncated octahedron by atomic coloring. Sites are instantiated with 50% randomly distributed CO coverage and zero oxygen coverage, which is set by the user in the `cover_system()` function. The system object is simply defined by the collection of sites and the function `identify_neigbors()`. Here, `identify_neighbors()` creates a neighbor list for each site based on distances in the associated `ase.Atoms` object. Sites are neighbors if the associated atoms are separated by the fcc nearest-neighbor distance $\left(\frac{a}{\sqrt{2}}\right)$.

Reaction energies are based on DFT calculations using $\overline{\text{CN}}$ as a descriptor (see the Appendix) as in Ref. 20. Adsorption energies, excluding adsorbate-adsorbate interactions, are stored as functions of $\overline{\text{CN}}$ as

$$E^{\text{ads}}_{\text{CO,Pt}}\left(\overline{\text{CN}}\right) = 1.403 - 0.252(\overline{\text{CN}} - 7.5), \tag{4}$$

$$E^{\text{ads}}_{\text{O,Pt}}\left(\overline{\text{CN}}\right) = 0.975 - 0.218(\overline{\text{CN}} - 7.5). \tag{5}$$

The energy barrier for CO_2 formation is calculated during the simulation through a Brønsted-Evans-Polanyi (BEP) relation in the CO and O adsorption energies

$$E_a = 2.95\,\text{eV} - 0.824\,\text{eV} \times \left(E^{\text{ads}}_{\text{O,Pt}} + E^{\text{ads}}_{\text{CO,Pt}}\right). \tag{6}$$

Adsorbate-adsorbate interactions are based on the nearest neighbor occupations, which are retrieved as: `system.get_neighbor_covs(i_site)`. The adsorbate-adsorbate interactions are stored in a 3 × 3 python-list

$$R = \begin{pmatrix} 0 & 0 & 0 \\ 0 & E_{\text{CO–CO}} & E_{\text{O–CO}} \\ 0 & E_{\text{CO–O}} & E_{\text{O–O}} \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0.19 & 0.30 \\ 0 & 0.30 & 0.32 \end{pmatrix} \text{eV}. \tag{7}$$

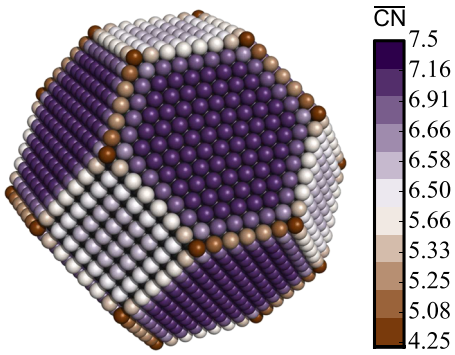

FIG. 4. Ontop sites on the truncated octahedron colored according to $\overline{\text{CN}}$.

Thus, the repulsive energy addition δE to species 1 (CO) on a site is

$$\delta E = \sum_n R_{1,c_n}. \tag{8}$$

Here c_n is the occupation on the nearest neighbor n, which is 0 for empty sites, 1 for CO, and 2 for O. The repulsions are added to (4) and (5) during the simulation, which also affects the barrier through (6). Diffusion barriers between isoenergetic sites are set to 0.08 eV for CO and 0.58 eV for O. These barriers are added to the difference in adsorption energies between the sites for thermodynamic consistency. To speed up simulation convergence, all diffusion barriers of CO are raised to a constant value of 0.45 eV. This value ensures convergence of the TOF.[20] Similar approaches have been applied and justified previously.[15–18,20,28,29]

For each reactive event, one needs to define the function `possible(system,i_site,i_other)` that returns `True` if the event presently is possible, the function `get_rate(system,i_site,i_other)` that returns the rate constant, and the function `do_event(system,i_site,i_other)` that modifies the occupation according to the reaction. To enhance readability, we will not repeat the input arguments to these functions in the following discussion.

In the present example, one event is defined for each possible reaction in the scheme (3). For the CO adsorption event, a class named `COAdsorption` is derived from the base class `EventBase`. The class needs a number of parameters for calculating the rate-constants: temperature (T), a constant CO pressure (p_{CO}), CO mass (m_{CO}), the site area (A_{site}), and the CO sticking coefficient (s_0). These parameters are stored on the `COAdsorption` object. The `possible()` function returns `True` if the site is empty. The `get_rate()` function returns the following collision theory rate-constant:

$$k^{\text{ads}}_i = \frac{A_{\text{site}} s_{0,i} p_{\text{CO}}}{\sqrt{2\pi m_{\text{CO}} k_{\text{B}} T}}, \tag{9}$$

where i labels the site. The sticking coefficient of CO was set to 0.9 for the facet sites and to 1 for edges and corners.

The function `do_event()` covers the site with species number 1 (CO). This is done by invoking the following:

```
system.sites[site_index].covered = 1.
```

The oxygen adsorption event class is defined in complete analogy to CO adsorption. The `possible()` function returns `True` if the two neighboring sites passed to the function are unoccupied. The `do_event()` function changes the occupation of these sites to 2 (O covered).

To implement the desorption events, similar classes are defined. However, the desorption rate-constants are calculated from the equilibrium constants and adsorption rate-constants. The `possible()` function returns true if the site is covered with 1 (CO) for CO desorption and two neighboring sites with 2 (O) for oxygen desorption. For CO desorption, `get_rate()` returns the following desorption rate-constant:

$$k^{\text{des}}_i = \frac{k^{\text{ads}}_i}{K_i}, \quad K_i = \exp\left(\frac{-\Delta G_i}{k_{\text{B}} T}\right), \tag{10}$$

where ΔG_i is the Gibbs free energy of adsorption, which depends on the adsorbate-adsorbate interactions. The enthalpic part of ΔG_i is given by (4) and (5). The entropic part of ΔG_i is implemented as the entropy difference between the gas-phase molecule in the ideal-gas approximation and the adsorbate in the harmonic approximation.

The CO_2 formation event is modeled as irreversible. The `possible()` function returns true if CO is on the site and O is on the neighbor site passed to the function. `get_rate()` returns a rate-constant derived from transition state theory,[30]

$$k_{i,j}^{\mathrm{TST}} = \frac{k_{\mathrm{B}}T}{h}\exp\left(\frac{-\Delta G_{i,j}^{\ddagger}}{k_{\mathrm{B}}T}\right), \tag{11}$$

where $\Delta G_{i,j}^{\ddagger}$ is the Gibbs-free energy barrier for the $CO^* + O^* \rightarrow CO_2(g) + 2*$ reaction with CO on site i and O on site j. The enthalpy is defined in (6), and the entropic part of the barrier is treated in the harmonic approximation. `do_event()` simply assigns 0 to both the site and neighbor site in question.

Events are also defined for diffusion of adsorbed CO and O. `possible()` returns `True` if an adsorbate covers the site in question and the neighbor site is free. The `get_rate()` function returns a transition state theory rate equivalent to (11).

Figures 5(a) and 5(b) show the CO and O coverage on the different sites as a function of time at 700 K. For CO, the edges and corners have the highest coverages close to 100%, whereas the coverages on the (100) and (111) facets are about 33% and 15%, respectively. For oxygen, (100) has the highest coverage of ca 20%, the (111) facets have about 10%, whereas edges and corners have coverages below 5%. The relative magnitudes of coverages are explained by the fact that corners have the highest reactivity, edges have the next highest reactivity, (100) and finally (111) sites have the lowest reactivity. The fact that edges and corners have small O coverages is owing to CO blocking. However, small fluctuations allow transient oxygen coverages on edges and corners. For both CO and O, there is an initial equilibration period, which in the present case corresponds to roughly half the simulation time. The CO coverage has a larger fluctuation amplitude and frequency as compared to oxygen. Figure 5(c) shows the simulated Turnover frequency (TOF) as a function of temperature. The TOF is calculated by logging the number of times the CO_2 formation event is performed after the initial equilibration. Error-bars stem from an average taken over 16 simulations. The TOF follows a light-off behavior where the value increases rapidly at about 800 K. The rapid increase is due to CO desorption, which enables dissociative oxygen adsorption and thus CO_2 formation. The present example illustrates how simulations over complex geometries can be performed by using an `ase.Atoms` object to define the sites and scaling relations to define the reaction energy landscape.

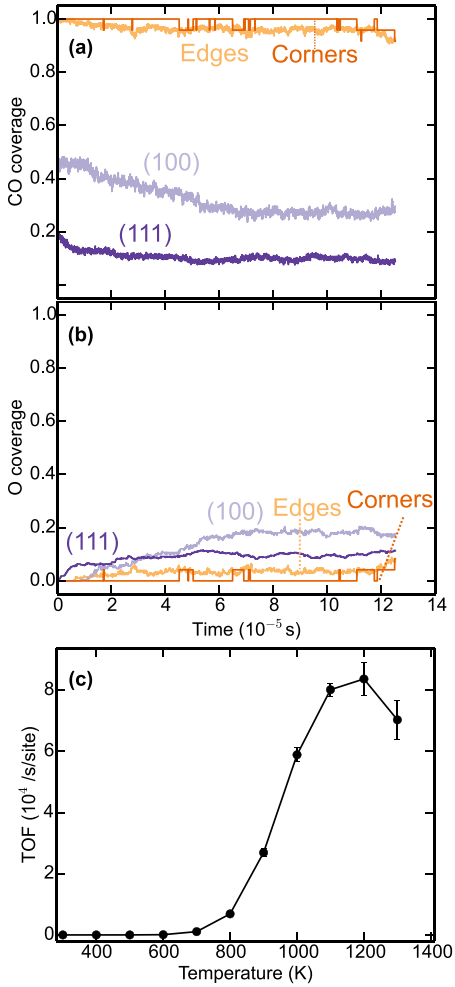

FIG. 5. (a) CO coverage and (b) O coverage of different site-types plotted as a function of time at 700 K. (c) Average turnover frequency of CO oxidation as a function of temperature. CO pressure: 2 mbar, O_2 pressure: 1 mbar.

B. Methane adsorption in zeolites

To demonstrate the generality of the present implementation, in this example, methane adsorption is modeled in a zeolite. The adsorption, desorption, and diffusion are modeled using the following scheme:

$$\begin{aligned} CH_4(g) + * &\leftrightarrow CH_4^*, \\ CH_4^* + * &\leftrightarrow * + CH_4^*. \end{aligned} \tag{12}$$

To simulate adsorption isotherms in a porous system such as a zeolite, the sites are conveniently put on a 3-dimensional grid of unit-cells. Such a grid is illustrated in Fig. 6, where there are four distinct types of cells (`stype`): 0—bulk, 1—facet, 2—edge, and 3—corner. The facets have one entry available for adsorption, the edges have two, and the corners have three entries. The `Site` object is assigned a position vector (i, j, k), which is used to identify the site-type. For example, for bulk sites, all indices are larger than zero and smaller than the number of sites-1. Non-bulk sites are exposed to the gas and are tagged with a custom variable `gas_site = True`. The `identify_neighbors()` function identifies neighbor cells as those with distance 1, which is calculated in the lattice-space by the metric

$$d = \sqrt{(i_2 - i_1)^2 + (j_2 - j_1)^2 + (k_2 - k_1)^2}, \tag{13}$$

where d is the distance between two cells at positions (i_2, j_2, k_2) and (i_1, j_1, k_1).

The reactive events are defined in the next step. For CH_4 adsorption, `possible()` returns `True` in empty non-bulk

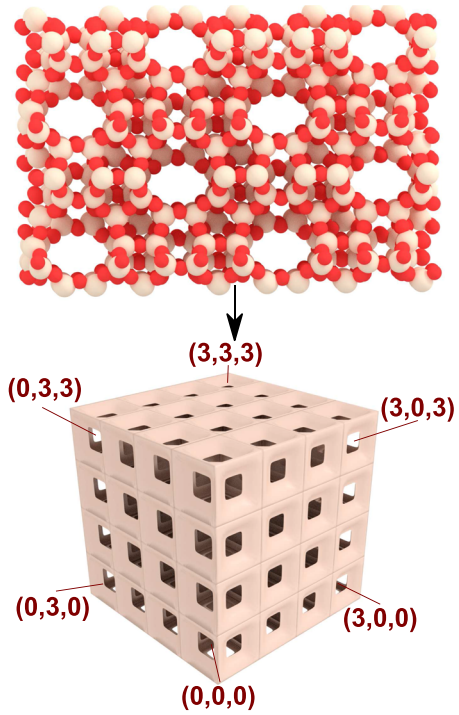

FIG. 6. Upper: Atomistic model of MFI zeolite channels. Lower: A cubic lattice model system with interconnected cells. Points indicate position in the lattice space.

cells (`gas_site = True`). `get_rate()` returns the collision theory rate-constant with constant CH_4 pressure as in (9), multiplied by the number of entries. `do_event()` simply changes `system.sites[i_site].covered` to 1. For CH_4 desorption, `possible()` returns `True` if the site is CH_4 covered and non-bulk. The `get_rate()` function returns the desorption rate-constant defined in (10) multiplied by the number of entries, and `do_event()` changes covered to 0.

The energy landscape is modeled by DFT calculations (see the Appendix). The CH_4 adsorption energy in silicalite of zeolite framework type MFI silicalite is found to be −0.38 eV, excluding zero point energy corrections. Barriers for CH_4 diffusion between cells are arbitrarily set to 0.35 eV. The entropy of CH_4 in the gas-phase is modeled in the ideal gas approximation and inside the framework it is approximated to be 2/3 of the gas-phase entropy.[31] The simulations are compared with the Langmuir model for the concentration,

$$\theta_{\mathrm{LM}} = \frac{K}{1+K}, \quad K = \exp\left(\frac{-\Delta G}{k_{\mathrm{B}}T}\right), \tag{14}$$

where ΔG is the Gibbs free energy change upon entering the zeolite from the gas-phase.

Figure 7 shows the simulated concentrations and the Langmuir result (14) as a function of temperature. At low temperatures, the predicted concentration is high and falls off as the temperature is increased. The concentrations predicted by the Langmuir model and the single unit-cell Monte Carlo are identical. However, the 20 × 20 × 20 system gives slightly higher concentrations. This we attribute to pore-blocking effects where methane has to find a free route out from the bulk, which has been discussed previously in the context of zeolite deactivation.[32] In reality, the pore-blocking effect is likely not pronounced for methane adsorption in MFI as two adsorbates could be present in the same cell. However, the present method is directly applicable to study pore-blocking phenomena in other systems. A conceptual difference between the kMC and the Langmuir model is configurational entropy. The kMC simulations include configurational entropy explicitly, whereas the Langmuir model is a mean-field theory that implements configurational entropy by assuming a random adsorbate distribution.

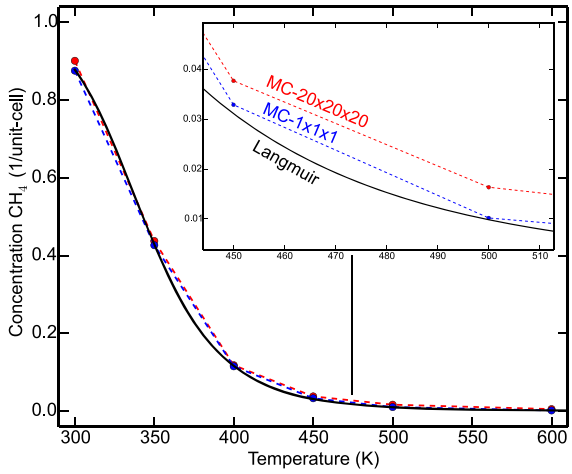

FIG. 7. Simulated methane concentration versus temperature at 1 mbar methane pressure: Langmuir model (black), Monte Carlo for 20 × 20 × 20 unit-cells (red), and Monte Carlo for 1 × 1 × 1 unit-cell (blue).

The present example illustrates that the object-oriented approach of MonteCoffee straightforwardly can simulate kinetics in a complex 3-dimensional pore network, such as a zeolite.

VI. CONCLUSION

The MonteCoffee simulation framework enables rapid development of customized kinetic Monte Carlo (kMC) simulations, which is highly suitable for simulations that require complex geometries. The code is a programmable application written in python, which provides a basic framework for performing kMC simulations. Here we presented the structure and outline of the code. CO oxidation over Pt nanoparticles with a descriptor-based energy landscape and CH_4 adsorption and diffusion inside zeolite pores were discussed as representative examples. MonteCoffee enables kMC simulations with full control over execution and rapid code development and thus it has the potential to considerably aid in the understanding of catalytic reaction kinetics.

ACKNOWLEDGMENTS

Financial support is acknowledged from Chalmers Excellence Initiative Nanoscience and Nanotechnology and the Swedish Research Council (2016-05234). The calculations were performed at PDC (Stockholm) and C3SE (Göteborg) via a SNIC grant. The Competence Centre for Catalysis (KCK) is hosted by Chalmers University of Technology and is financially supported by the Swedish Energy Agency and the member companies AB Volvo, ECAPS AB, Johnson Matthey AB,

Preem AB, Scania CV AB, Umicore Denmark ApS, and Volvo Car Corporation AB.

The modules are available at https://gitlab.com/ChemPhysChalmers/MonteCoffee.

APPENDIX: DENSITY FUNCTIONAL THEORY

Reaction energies were calculated by density functional theory with the Vienna *ab initio* simulation (VASP) package.[33–35] A plane-wave basis with a kinetic cutoff of 450 eV was used. Core-valence electronic interactions were treated in the Projector-Augmented Wave (PAW) scheme with valence electrons: Pt(10), Si(4), O(6), C(4), and H(1). Ionic relaxations were performed in ASE with the BFGS-Linesearch algorithm using a force convergence criterion of 0.05 eV/Å. Gas-phase molecules were relaxed in cells of at least (22 Å × 22 Å × 22 Å). Vibrational energies were calculated in the harmonic approximation with finite differences of displacement 0.01 Å. Spin-polarization was applied only to O_2 in the gas-phase, which was treated in the triplet spin-state. Energy barriers were evaluated using the Nudged Elastic Band[36] (NEB) function from the Variational Transition State Theory (VTST) tools[37] with seven images. Initial interpolations between the initial and final state were done using the image dependent pair potential.[38]

The reaction energies for CO oxidation over nanoparticles were calculated in Ref. 20 using the Revised Perdew-Burke-Ernzerhof (RPBE)[39] exchange-correlation functional. The Pt lattice-constant was calculated in the fcc unit cell to be 4.00 Å using a (12 × 12 × 12) k-point grid. Model surfaces were treated as four-layer slabs in at least (2 × 2) cells using (6 × 6 × 1) k-points. A 12 Å vacuum was introduced perpendicular to the slab separating periodic images.

In the zeolite-example, the Bayesian Error Estimation Functional with van der Waals correlation (BEEF-vdW)[40] was used to model exchange and correlation. The adsorption energy of CH_4 was found by local relaxation initiated from multiple positions of the molecule.

[1] J. K. Nørskov, T. Bligaard, J. Rossmeisl, and C. H. Christensen, Nat. Chem. **1**, 37 (2009).
[2] M. J. Hoffmann, M. Scheffler, and K. Reuter, ACS Catal. **5**, 1199 (2015).
[3] M. Stamatakis and D. G. Vlachos, ACS Catal. **2**, 2648 (2012).
[4] D. T. Gillespie, J. Comput. Phys. **22**, 403 (1976).
[5] K. A. Fichthorn and W. H. Weinberg, J. Chem. Phys. **95**, 1090 (1991).
[6] M. J. Hoffmann, S. Matera, and K. Reuter, Comput. Phys. Commun. **185**, 2138 (2014).
[7] J. J. Lukkien, "Carlos," accessed online 24 January 2018, https://carlos.win.tue.nl/.
[8] S. S. Plimpton, A. Thompson, and A. Slepoy, "Spparks kinetic monte carlo simulator," accessed online 24 January 2018, http://spparks.sandia.gov/.
[9] M. Leetmaa and N. V. Skorodumova, Comput. Phys. Commun. **185**, 2340 (2014).
[10] M. Leetmaa and N. V. Skorodumova, Comput. Phys. Commun. **196**, 611 (2015).
[11] F. M. Kunz, L. Kuhn, and O. Deutschmann, J. Chem. Phys. **143**, 044108 (2015).
[12] M. Stamatakis and D. G. Vlachos, J. Chem. Phys. **134**, 214115 (2011).
[13] P. F. Dubois, Comput. Phys. **8**, 70 (1994).
[14] S. R. Bahn and K. W. Jacobsen, Comput. Sci. Eng. **4**, 56 (2002).
[15] A. P. J. Jansen, *An Introduction to Kinetic Monte Carlo Simulations of Surface Reactions* (Springer, Berlin, 2012), pp. 11–12; 38; 53; 162–163.
[16] A. Chatterjee and A. F. Voter, J. Chem. Phys. **132**, 194101 (2010).
[17] M. Stamatakis and D. G. Vlachos, Comput. Chem. Eng. **35**, 2602 (2011).
[18] E. C. Dybeck, C. P. Plaisance, and M. Neurock, J. Chem. Theory Comput. **13**, 1525 (2017).
[19] See https://docs.python.org/2/tutorial/classes.html for a description of object oriented programming in python; accessed online 15 May 2018.
[20] M. Jørgensen and H. Grönbeck, ACS Catal. **7**, 5054 (2017).
[21] M. Jørgensen and H. Grönbeck, Angew. Chem., Int. Ed. **57**, 5086 (2018).
[22] T. Nilsson Pingel, M. Jørgensen, A. Yankovich, H. Grönbeck, and E. Olsson, Nat. Commun. **9**, 2722 (2018).
[23] P. Hohenberg and W. Kohn, Phys. Rev. **136**, B864 (1964).
[24] W. Kohn and L. J. Sham, Phys. Rev. **140**, A1133 (1965).
[25] See https://wiki.fysik.dtu.dk/ase/ase/atoms.html for a description of the ase atoms object; accessed online 15 June 2018.
[26] F. Calle-Vallejo, J. I. Martínez, J. M. García-Lastra, P. Sautet, and D. Loffreda, Angew. Chem., Int. Ed. **53**, 8316 (2014).
[27] F. Calle-Vallejo, D. Loffreda, M. T. M. Koper, and P. Sautet, Nat. Chem. **7**, 403 (2015).
[28] B. Puchala, M. L. Falk, and K. Garikipati, J. Chem. Phys. **132**, 134104 (2010).
[29] K. A. Fichthorn and Y. Lin, J. Chem. Phys. **138**, 164104 (2013).
[30] H. Eyring, Chem. Rev. **17**, 65 (1935).
[31] C. Paolucci, A. A. Parekh, I. Khurana, J. R. D. Iorio, H. Li, J. D. Albarracin Caballero, A. J. Shih, T. Anggara, W. N. Delgass, J. T. Miller, F. H. Ribeiro, R. Gounder, and W. F. Schneider, J. Am. Chem. Soc. **138**, 6028 (2016).
[32] D. Cai, Y. Ma, Y. Hou, Y. Cui, Z. Jia, and C. Zhang, Catal. Sci. Technol. **7**, 2440 (2017).
[33] G. Kresse and J. Furthmuller, Phys. Rev. B **54**, 11169 (1996).
[34] G. Kresse and J. Furthmuller, Comput. Mater. Sci. **6**, 15 (1996).
[35] G. Kresse and J. Hafner, Phys. Rev. B **47**, 558 (1993).
[36] G. Henkelman, B. P. Uberuaga, and H. Jónsson, J. Chem. Phys. **113**, 9901 (2000).
[37] See http://theory.cm.utexas.edu for "Henkelman Group at the University of Texas at Austin Home Page" (accessed November 16, 2015).
[38] S. Smidstrup, A. Pedersen, K. Stokbro, and H. Jónsson, J. Chem. Phys. **140**, 214106 (2014).
[39] B. Hammer, L. B. Hansen, and J. K. Nørskov, Phys. Rev. B **59**, 7413 (1999).
[40] J. Wellendorff, K. T. Lundgaard, A. Møgelhøj, V. Petzold, D. D. Landis, J. K. Nørskov, T. Bligaard, and K. W. Jacobsen, Phys. Rev. B **85**, 235149 (2012).