kmos Documentation

Release 0.3.16

Max J. Hoffmann

January 28, 2016

Contents

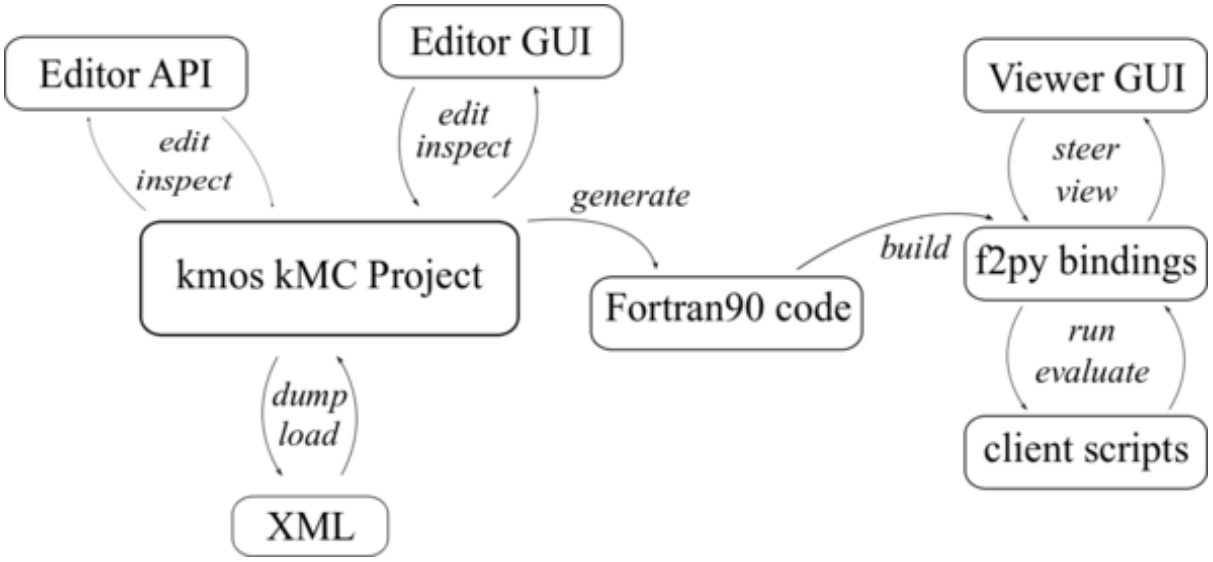

Fig. 1: Things you can do with kmos.

kmos is a vigorous attempt to make (lattice) kMC modelling more accessible.

kmos is designed for and by kMC model developers. As of this writing there is no standardized way to develop kMC models, thus there is no standardized way to use kmos. kmos can be an Editor, an API, a viewer. However all in all kmos wants to save time filled with repetitive labor and enlarge your stride.

Not sure how to begin? Start with the *API tutorial*.

CHAPTER 1

Tutorials

1.1 Installation

1.1.1 Introduction

Feature overview

With kmos you can:

- easily create and modify kMC models through GUI
- store and exchange kMC models through XML
- generate fast, platform independent, self-contained code [1]
- run kMC models through GUI or python bindings

kmos has been developed in the context of first-principles based modelling of surface chemical reactions but might be of help for other types of kMC models as well.

kmos' goal is to significantly reduce the time you need to implement and run a lattice kmc simulation. However it can not help you plan the model.

kmos can be invoked directly from the command line in one of the following ways:

```
kmos [help] (all|benchmark|build|edit|export|export-settings|help|import|rebuild|run|view) [optio
```

or it may be used as an API via the kmos module.

1.1.2 Installation on Ubuntu Linux

You can fetch the current version of kmos using *git*

```
git clone http://www.github.com/mhoffman/kmos
```

and install it using *setuptools*

```
./setup.py install [--user]
```

or if you have pip run

```
pip install python-kmos --upgrade [--user]
```

To use the core functionality (programmatic model setup, code generation, model execution) kmos has a fairly modest depedency foot-print. You will need

[1] The source code is generated in Fortran90, written in a modular fashion. Python bindings are generated using f2py.

```
python-numpy, a Fortran compiler, python-lxml
```

In order to watch the model run on screen you will additionally need

```
python-matplotlib, python-ase
```

Finally in order to use all features, in particular the GUI model editor of kmos you have to install a number of dependencies. This should not be very difficult on a recent Linux distribution with package management. So on Ubuntu it suffices to call:

```
sudo apt-get install gazpacho gfortran python-dev \
                     python-glade2 python-kiwi python-lxml \
                     python-matplotlib python-numpy \
                     python-pygoocanvas
```

and if you haven't already installed it, one way to fetch the atomic simulation environment (ASE) is currently to

```
sudo add-apt-repository ppa:campos-dev/campos
sudo apt-get update
sudo apt-get install python-ase
```

Or go to their website to fetch the latest version.

Unfortunately Debian/Ubuntu have discontinued maintaining the gazpacho package which I find very unfortunate since it eased gtk GUI building a lot and I haven't found a simple transition path (simple as in one reliable conversion script and two changed import lines) towards gtkbuilder. Therefore for the moment I can only suggest to fetch the latest old package from e.g. Debian Packages and install it manually with

```
sudo dpkg -i gazpacho_*.deb
```

If you think this dependency list hurts. Yes, it does! And I am happy about any suggestions how to minimize it. However one should note these dependencies are only required for the model development. Running a model has virtually no dependencies except for a Fortran compiler.

To ease the installation further on Ubuntu one can simply run:

```
kmos-install-dependencies-ubuntu
```

1.1.3 Installation on openSUSE 12.1 Linux

On a recent openSUSE some dependencies are distributed a little different but nevertheless doable. We start by install some package from the repositories

```
sudo zypper install libgfortran46, python-lxml, python-matplotlib, \
                    python-numpy, python-numpy-devel, python-goocanvas,
                    python-imaging
```

And two more packages SUSE packages have to be fetched from the openSUSE build service

- gazpacho
- python-kiwi

For each one just download the *.tar.bz2 files. Unpack them and inside run

```
python setup.py install
```

In the same vein you can install ASE. Download a recent version from the DTU website unzip it and install it with

```
python setup.py install
```

1.1.4 Installation on openSUSE 13.1 Linux

In order to use the editor GUI you will want to install python-kiwi (not KIWI) and right now you can find a recent build here .

1.1.5 Installation on Mac OS X 10.10 or above

There is more than one way to get required dependencies. I have tested MacPorts and worked quite well.

1. **Get MacPorts** Search for MacPorts online, you'll need to install Xcode in the process
2. Install Python, lxml, numpy, ipython, ASE, gcc48. I assume you are using Python 2.7. kmos has not been thoroughly tested with Python 3.X, yet, but should not be too hard.

   Having MacPorts this can be as simple as

```
sudo ports install -v py27-ipython
sudo port select --set ipython ipython ipython27

sudo port install gcc48
sudo port select --set gcc mp-gcc48 # need to that f2py finds a compiler

sudo port install py27-readline
sudo port install py27-goocanvas
sudo port install py27-lxml
sudo port install kiwi
# possibly more ...

# if you install these package manually, skip pip :-)
sudo port install py27-pip
sudo port select --set pip pip27

pip install python-ase --user
pip install python-kmos --user
```

1.1.6 Installation on windoze 7

In order for kmos to work in a recent windoze we need a number of programs.

1. **Python** If you have no python previously installed you should try Enthought Python Distribution (EPD) in its free version since it already comes with a number of useful libraries such a numpy, scipy, ipython and matplotlib.

   Otherwise you can simply download Python from python.org and this installation has been successfully tested using python 2.7.
2. **numpy** Fetch it for *your version* of python from sourceforge's Numpy site and install it. [Not needed with EPD]
3. **MinGW** provides free Fortran and C compilers and can be obtained from the sourceforge's MinGW site . Make sure you make a tick for the Fortran and the C compiler.
4. **pyGTK** is needed for the GUI frontend so fetch the all-in-one bundle installer and install most of it.
5. **lxml** is an awesome library to process xml files, which has unfortunately not fully found its way into the standard library. As of this writing the latest version with prebuilt binaries is lxml 2.2.8 and installation works without troubles.
6. **ASE** is needed for the representation of atoms in the frontend. So download the latest from the DTU website and install it. This has to be installed using e.g. the powershell. So after unpacking it, fire up the powershell, cd to the directory and run

```
python setup.py install
```

in there. Note that there is currently a slight glitch in the *setup.py* script on windoze, so open *setup.py* in a text editor and find the line saying

```
version = ...
```

comment out the lines above it and hard-code the current version number.

7. **kmos** is finally what we are after, so download the latest version from github and install it in the same way as you installed **ASE**.

There are probably a number of small changes you have to make which are not described in this document. Please post questions and comments in the issues forum .

1.1.7 Installing JANAF Thermochemical Tables

You can conveniently use gas phase chemical potentials inserted in rate constant expressions using JANAF Thermochemical Tables. A couple of molecules are automatically supported. If you need support for more gas-phase species, drop me a line.

The tabulated values are not distributed since the terms of distribution do not permit this. Fortunately manual installation is easy. Just create a directory called *janaf_data* anywhere on your python path. To see the directories on your python path run

```
python -c"import sys; print(sys.path)"
```

Inside the *janaf_data* directory has to be a file named *__init__.py*, so that python recognizes it as a module

```
touch __init__.py
```

Then copy all needed data files from the NIST website in the tab-delimited text format to the *janaf_data* directory. To download the ASCII file, search for your molecule. In the results page click on 'view' under 'JANAF Table' and click on 'Download table in tab-delimited text format.' at the bottom of that page.

Todo

test installation on other platforms

1.2 Screenshots

1.2.1 The Editor

1.2.2 The Runtime View

1.3 A first kMC Model–the API way

In general there are two interfaces to *defining* a new model: A GUI and an API. While the GUI can be quite nice especially for beginners, it turns out that the API is better maintained simply because ... well, maintaing a GUI is a lot more work.

So we will start by learning how to setup the model using the API which will turn out not to be hard at all. It is knowing howto do this will also pay-off especially if you starting tinkering with your existing models and make little changes here and there.

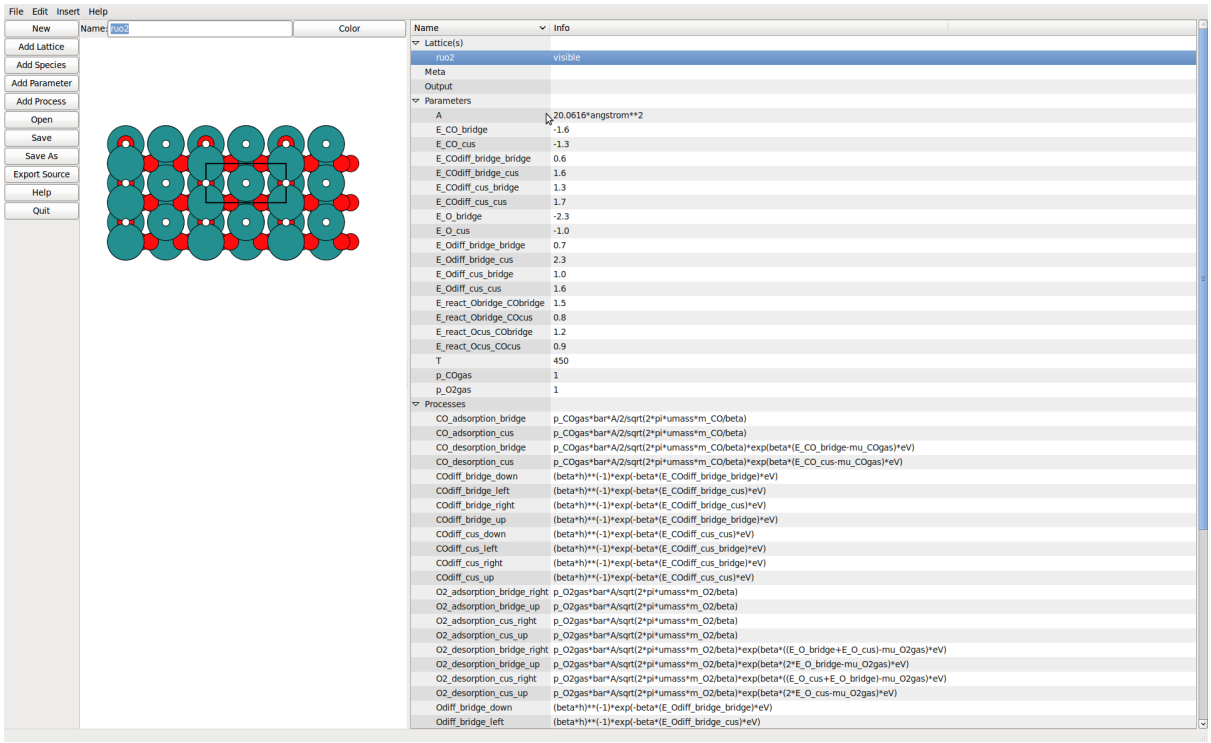

Fig. 1.1: The lattice view allows to define sites by simple pointing.

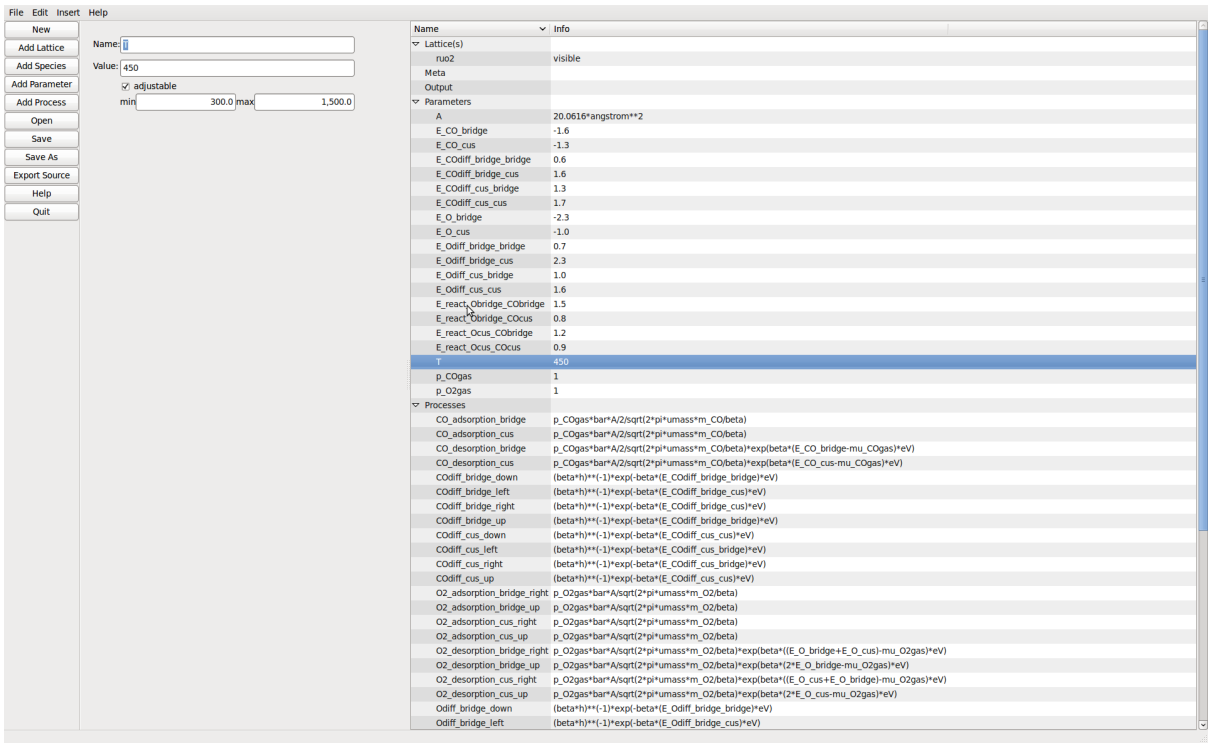

Fig. 1.2: Model parameters can be defined including ranges to vary them over in the runtime viewer.

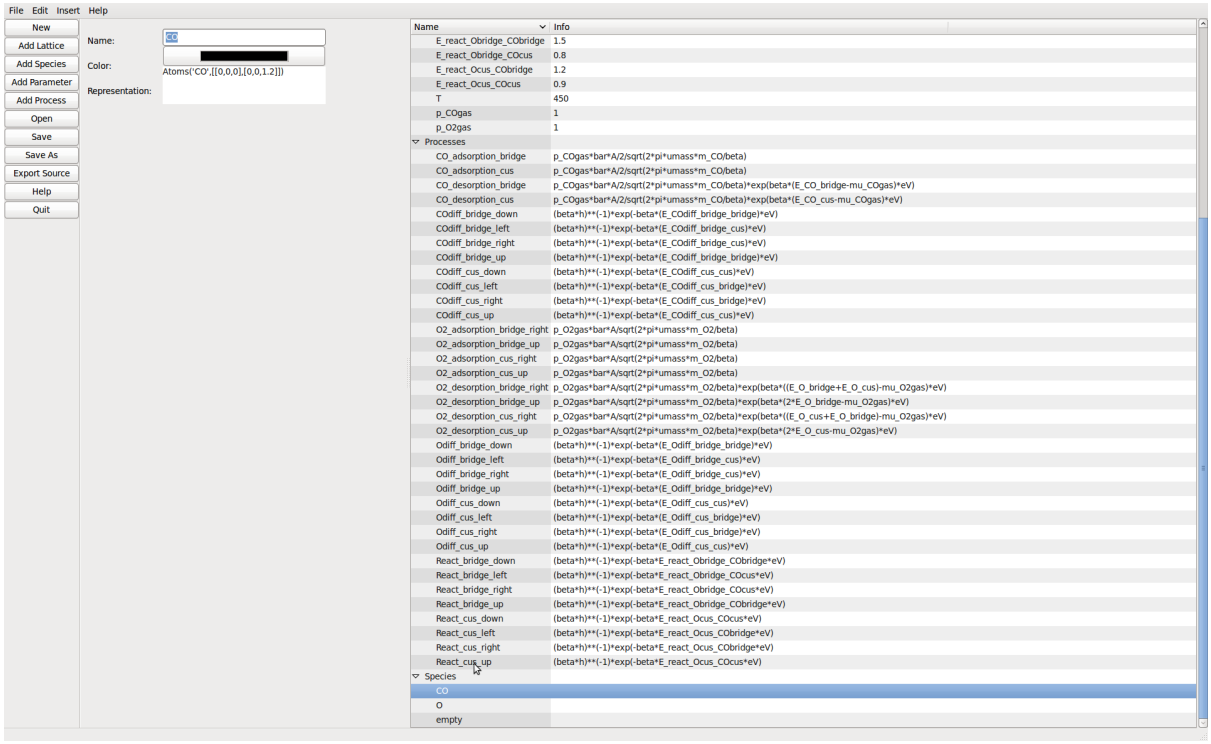

Fig. 1.3: Species can be added here. The color is used to represent them in the 2D editor view. The string is an ASE atoms constructor for display at runtime.

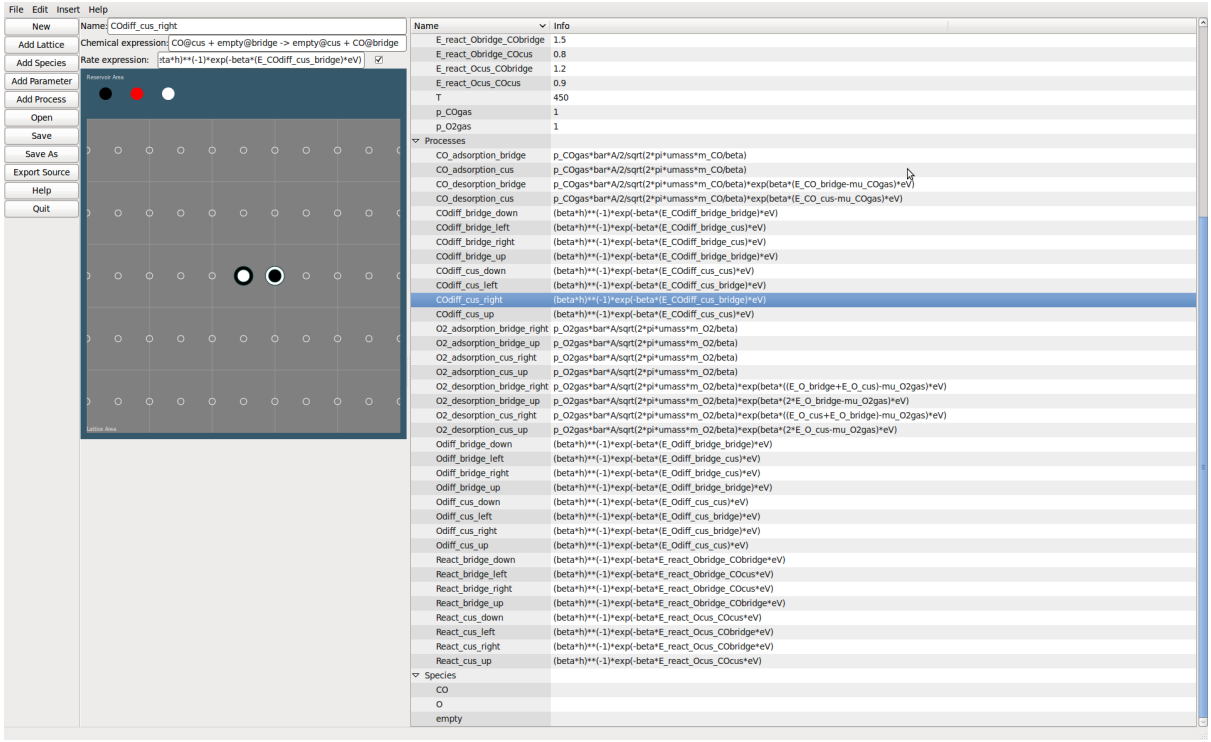

Fig. 1.4: Processes can be added by point and click or by entering a chemical expression.

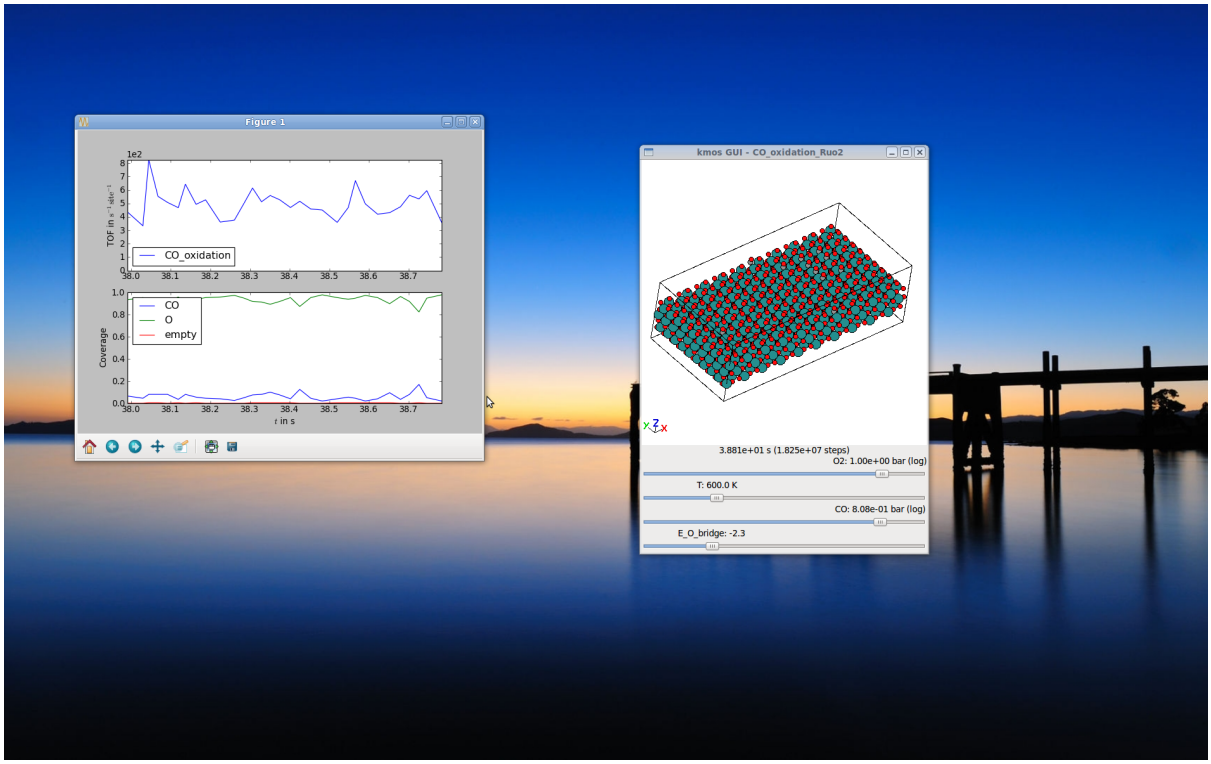

Fig. 1.5: The compiled module can be run and watched in realtime. When parameters are changed this is immediately reflected in the rate constants.

1.3.1 Construct the model

We start by making the necessary import statements (in *python* or better *ipython*):

```
from kmos.types import *
from kmos.io import *
import numpy as np
```

which imports all classes that make up a kMC project. The functions from *kmos.io* will only be needed at the end to save the project or to export compilable code.

The example sketched out here leads you to a kMC model for CO adsorption and desorption on Pd(100) including a simple lateral interaction. Granted this hardly excites surface scientists but we need to start somewhere, right?

First you should instantiate a new project and fill in meta information

```
pt = Project()
pt.set_meta(author = 'Your Name',
            email = 'your.name@server.com',
            model_name = 'MyFirstModel',
            model_dimension = 2,)
```

Next you add some species or states. Note that whichever species you add first is the default species with which all sites in the system will be initialized. Of course this can be changed later

For surface science simulations it is useful to define an *empty* state, so we add

```
pt.add_species(name='empty')
```

and some surface species. Given you want to simulate CO adsorption and desorption on a single crystal surface you would say

```
pt.add_species(name='CO',
               representation="Atoms('CO',[[0,0,0],[0,0,1.2]])")
```

where the string passed as *representation* is a string representing a CO molecule which can be evaluated in ASE namespace.

Once you have all species declared is a good time to think about the geometry. To keep it simple we will stick with a simple-cubic lattice in 2D which could for example represent the (100) surface of a fcc crystal with only one adsorption site per unit cell. You start by giving your layer a name

```
layer = pt.add_layer(name='simple_cubic')
```

and adding a site

```
layer.sites.append(Site(name='hollow', pos='0.5 0.5 0.5',
                        default_species='empty'))
```

Where *pos* is given in fractional coordinates, so this site will be in the center of the unit cell.

Simple, huh? Now you wonder where all the rest of the geometry went? For a simple reason: the geometric location of a site is meaningless from a kMC point of view. In order to solve the master equation none of the numerical coordinates of any lattice sites matter since the master equation is only defined in terms of states and transition between these. However to allow a graphical representation of the simulation one can add geometry as you have already done for the site. You set the size of the unit cell via

```
pt.lattice.cell = np.diag([3.5, 3.5, 10])
```

which are prototypical dimensions for a single-crystal surface in Angstrom.

Ok, let us see what we managed so far: you have a *lattice* with a *site* that can be either *empty* or occupied with *CO*.

1.3.2 Populate process list and parameter list

The remaining work is to populate the *process list* and the *parameter list*. The parameter list defines the parameters that can be used in the expressions of the rate constants. In principle one could do without the parameter list and simply hard code all parameters in the process list, however one looses some nifty functionality like easily changing parameters on-the-fly or even interactively.

A second benefit is that you achieve a clear separation of the kinetic model from the barrier input, which usually has a different origin.

In practice filling the parameter list and the process list is often an iterative process, however since we have a fairly short list, we can try to set all parameters at once.

First of all you want to define the external parameters to which our model is coupled. Here we use the temperature and the CO partial pressure:

```
pt.add_parameter(name='T', value=600., adjustable=True, min=400, max=800)
pt.add_parameter(name='p_CO', value=1., adjustable=True, min=1e-10, max=1.e2)
```

You can also set a default value and a minimum and maximum value set defines how the scrollbars a behave later in the runtime GUI.

To describe the adsorption rate constant you will need the area of the unit cell:

```
pt.add_parameter(name='A', value='(3.5*angstrom)**2')
```

Last but not least you need a binding energy of the particle on the surface. Since without further ado we have no value for the gas phase chemical potential, we'll just call it deltaG and keep it adjustable

```
pt.add_parameter(name='deltaG', value='-0.5', adjustable=True,
                            min=-1.3, max=0.3)
```

To define processes we first need a coordinate [3]

[3] The description of coordinates follows the simple syntax of the coordinate syntax and the *topic guide* explains how that works.

```
coord = pt.lattice.generate_coord('hollow.(0,0,0).simple_cubic')
```

Then you need to have at least two processes. A process or elementary step in kMC means that a certain local configuration must be given so that something can happen at a certain rate constant. In the framework here this is phrased in terms of 'conditions' and 'actions'. [2] So for example an adsorption requires at least one site to be empty (condition). Then this site can be occupied by CO (action) with a rate constant. Written down in code this looks as follows

```
pt.add_process(name='CO_adsorption',
               conditions=[Condition(coord=coord, species='empty')],
               actions=[Action(coord=coord, species='CO')],
               rate_constant='p_CO*bar*A/sqrt(2*pi*umass*m_CO/beta)')
```

Note: In order to ensure correct functioning of the kmos kMC solver every action should have a corresponding condition for the same coordinate.

Now you might wonder, how come we can simply use m_CO and beta and such. Well, that is because the evaluator will to some trickery to resolve such terms. So beta will be first be translated into 1/(kboltzmann*T) and as long as you have set a parameter *T* before, this will go through. Same is true for m_CO, here the atomic masses are looked up and added. Note that we need conversion factors of *bar* and *umass*.

Then the desorption process is almost the same, except the reverse:

```
pt.add_process(name='CO_desorption',
               conditions=[Condition(coord=coord, species='CO')],
               actions=[Action(coord=coord, species='empty')],
               rate_constant='p_CO*bar*A/sqrt(2*pi*umass*m_CO/beta)*exp(beta*deltaG*eV)')
```

To reduce typing, kmos also knows a shorthand notation for processes. In order to produce the same process you could also type

```
pt.parse_process('CO_desorption; CO@hollow->empty@hollow ; p_CO*bar*A/sqrt(2*pi*umass*m_CO/beta)*e
```

and since any non-existing on either the left or the right side of the -> symbol is replaced by a corresponding term with the *default_species* (in this case *empty*) you could as well type

```
pt.parse_process('CO_desorption; CO@hollow->; p_CO*bar*A/sqrt(2*pi*umass*m_CO/beta)*exp(beta*delta
```

and to make it even shorter you can parse and add the process on one line

```
pt.parse_and_add_process('CO_desorption; CO@hollow->; p_CO*bar*A/sqrt(2*pi*umass*m_CO/beta)*exp(be
```

In order to add processes on more than one site possible spanning across unit cells, there is a shorthand as well. The full-fledged syntax for each coordinate is

```
name.offset.lattice
```

check *Manual generation* for details.

1.3.3 Export, save, compile

Next, it's a good idea to save your work

```
pt.filename = 'myfirst_kmc.xml'
pt.save()
```

Now is the time to leave the python shell. In the current directory you should see a *myfirst_kmc.xml*. This XML file contains the full definition of your model and you can create the source code and binaries in just one line. So on the command line in the same directory as the XML file you run

[2] You will have to describe all processes in terms of *conditions* and *actions* and you find a more complete description in the *topic guide* to the process description syntax.

```
kmos export myfirst_kmc.xml
```

or alternatively if you are still on the ipython shell and don't like to quit it you can use the API hook of the command line interface like

```
import kmos.cli
kmos.cli.main('export myfirst_kmc.xml')
```

Make sure this finishes gracefully without any line containining an error.

If you now *cd* to that folder *myfirst_kmc* and run:

```
kmos view
```

... and dada! Your first running kMC model right there!

If you wonder why the CO molecules are basically just dangling there in mid-air that is because you have no background setup, yet. Choose a transition metal of your choice and add it to the lattice setup for extra credit :-).

Wondering where to go from here? If the work-flow makes complete sense, you have a specific model in mind, and just need some more idioms to implement it I suggest you take a look at the examples folder. for some hints. To learn more about the kmos approach and methods you should into *topic guides*.

1.3.4 Taking it home

Despite its simplicity you have now seen all elements needed to implement a kMC model and hopefully gotten a first feeling for the workflow.

Todo

describe modelling more complicated structures and e.g. boundary conditions

1.4 Running the Model–the GUI way

After successfully exporting and compiling a model you get two files: kmc_model.so and kmc_settings.py. These two files are really all you need for simulations. So a simple way to view the model is the

```
kmos view
```

command from the command line. For this two work you need to be in the same directory as these two file (more precisely these two files need to be in the python import path) and you should see an instance of your model running. This feature can be quite useful to quickly obtain an intuitive understanding of the model at hand. A lot of settings can be changed through the kmc_settings.py such as rate constant or parameters. To be even more interactive you can set a parameter to be adjustable. This can happen either in the generating XML file or directly in the kmc_settings.py. Also make sure to set sensible minimum and maximum values.

1.5 How To Prepare a Model and Run It Interactively

If you want to prepare a model in a certain way (parameters, size, configuration) and then run it interactively from there, there is in easy way, too. Just write a little python script. The with-statement is nice because it takes care of the correct allocation and deallocation

```
#!/usr/bin/env python

from kmos.run import KMC_Model
from kmos.view import main
```

```
with KMC_Model(print_rates=False, banner=False) as model:
  model.settings.simulation_size = 5

with KMC_Model(print_rates=False, banner=False) as model:
    model.do_steps(int(1e7))
    model.double()
    model.double()
    # one or more changes to the model
    # ...
    main(model)
```

Or you can use the hook in the *kmc_settings.py* called *setup_model*. This function will be invoked at startup every time you call

```
kmos view, run, or benchmark
```

Though it can easily get overwritten, when exporting or rebuilding. To minimize this risk, you e.g. place the *setup_model* function in a separate file called *setup_model.py* and insert into kmc_settings.py

```
from setup_model import setup_model
```

Next time you overwrite *kmc_settings.py* you just need to add this line again.

1.6 Running the Model–the API way

In order to analyze a model in quantitatively it is more practical to write small client scripts that directly talk to the runtime API. As time passes and more of these scripts are written over and over some standard functionality will likely be integrated into the runtime API. For starters a simple script could look as follows

```
#!/usr/bin/env python

from kmos.run import KMC_Model

model = KMC_Model()
```

An alternative way that gets you started fast it to run

```
kmos shell
```

and just interact directly with *model*. It is often a good idea to

```
%logstart some_scriptname.py
```

first in the IPython command to save what you have typed for later use.

As you can see by default the model prints a disclaimer and all rate constants, which can each be turned off by instantiating

```
model = KMC_Model(print_rates=False, banner=False)
```

The most important method is of course how to run the model, which you can do by saying

```
model.do_steps(100000)
```

which would run the model by 100,000 kMC steps.

Let's say you want to change the temperature and a partial pressure of the model you could type

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

and all rate constants are instantly updated. In order get a quick overview of the current settings you can issue e.g.

```
print(model.parameters)
print(model.rate_constants)
```

or just

```
print(model)
```

Now an instantiated und configured model has mainly two functions: run kMC steps and report its current configuration.

To analyze the current state you may use

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

Note: If you want to fetch data from the current state without actually visualizing the geometry can speed up the get_atoms() call using

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

This will return an ASE atoms object of the current system, but it also contains some additional data piggy-backed such as

```
model.get_occupation_header()
atoms.occupation

model.get_tof_header()
atoms.tof_data


atoms.kmc_time
atoms.kmc_step
```

These quantities are often sufficient when running and simulating a catalyst surface, but of course the model could be expanded to more observables. The Fortran modules *base*, *lattice*, and *proclist* are atttributes of the model instance so, please feel free to explore the model instance e.g. using ipython and

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

etc..

The *occupation* is a 2-dimensional array which contains the *occupation* for each surface *site* divided by the number of unit cell. The first slot denotes the species and the second slot denotes the surface site, i.e.

```
occupation = model.get_atoms().occupation
occupation[species, site-1]
```

So given there is a *hydrogen* species in the model, the occupation of *hydrogen* across all site type can be accessed like

```
hydrogen_occupation = occupation[model.proclist.hydrogen]
```

To access the coverage of one surface site, we have to remember to subtract 1, when using the the builtin constants, like so

```
hollow_occupation = occupation[:, model.lattice.hollow-1]
```

Lastly it is important to call

```
model.deallocate()
```

once the simulation if finished as this frees the memory allocated by the Fortan modules. This is particularly necessary if you want to run more than one simulation in one script.

1.7 Generate Grids of Sampled Data

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 *ModelRunner* to work with

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

class ScanKinetics(ModelRunner):
    p_O2gas = PressureParameter(1)
    T = TemperatureParameter(600)
    p_COgas = PressureParameter(min=1, max=10, steps=40)


ScanKinetics().run(init_steps=1e8, sample_steps=1e8, cores=4)
```

This script generates data points over the specified range(s). The temperature parameters is uniform grids over 1/T and the pressure parameters is uniform over log(p). The script can be run synchronously over many cores as long as the cores can access the same file system. You have to test whether the steps before sampling (*init_steps*) as well as the batch size (*sample_steps*) is sufficient.

1.8 Manipulating the Model at Runtime

It is quite easy to change not only model parameters but also the configuration at runtime. For instance if one would like to prepare a surface with a certain configuration or pattern.

Given you instantiated a *model* instance a site occupation can be changed by calling

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

However if changing many sites at once this is quite inefficient, since each *put* call, adjusts the book-keeping database. To circumvent this you can use the *_put* method, like so

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

though at the end one must not forget to call *_adjust_database()* before executing any next step or the database of available processes is inaccurate and the model instance will crash soon.

You can also get or set the whole configuration of the lattice at once using

```
config = model._get_configuration()
# possible change config
model._set_configuration(config)
```

1.9 Running models in parallel

Due to the global clock in kMC there seems to be no simple and efficient way to parallelize a kMC program. kmos certainly cannot parallelize a single system over processors. However one can run several kmos instances in parallel which might accelerate sampling or efficiently check for steady state conditions.

However in many applications it is still useful to run several models seperately at once, for example to scan some set of parameters one a multicore computer. This kind of problem can be considered *embarrasingly parallel* since it requires no communication between the runs.

This is made very simple through the *multiprocessing* module, which is in the Python standard library since version 2.6. For older versions this needs to be *downloaded* <*http://pypi.python.org/pypi/multiprocessing/*> and installed manually. The latter is pretty straightforward.

Then besides *kmos* we need to import *multiprocessing*

```
from multiprocessing import Process
from numpy import linspace
from kmos.run import KMC_Model
```

and let's say you wanted to scan a range of temperature, while keeping all other parameteres constant. You first define a function, that takes a set of temperatures and runs the simulation for each

```
def run_temperatures(temperatures):
    for T in temperatures:
        model = KMC_Model()
        model.parameters.T = T
        model.do_steps(100000)

        # do some evaluation

        model.deallocate()
```

In order to split our full range of input parameters, we can use a utility function

```
from kmos.utils import split_sequence
```

All that is left to do, is to define the input parameters, split the list and start subprocesses for each sublist

```
if __name__ == '__main__':
    temperatures = linspace(300, 600, 50)
    nproc = 8
    for temperatures in split_sequence(temperatures, nproc):
        p = Process(target=run_temperatures, args=(temperatures, ))
        p.start()
```

1.10 Development

Contributions of any sort are of course quite welcome. Patches and comments are ideally sent in form of email, pull request, or github issues.

To make synergizing a most pleasing experience I suggest you use git, nose, pep8, and pylint

```
sudo apt-get install git python-nose pep8 pylint
```

When sending a patch please make sure the nose tests pass, i.e. run from the top project directory

```
nosetests
```

Furthermore run

```
pep8 <filename>
```

Thank you.

CHAPTER 2

Topic Guides

The conceptual parts of this topic guide predate the kmos paper (arXiv). Please refer to the paper for a thorough background on kMC and lattice kMC on crystal surfaces. The more technical parts stated below might still be useful for using kmos.

2.1 The Concept of Kinetic Monte Carlo

2.1.1 Why use Kinetic Monte Carlo?

There is a class of systems in nature for which the spatiotemporal evolution can be described using a master type of equation. While chemical reactions at surfaces is one of them, it is not limited to those.

The master equation imposes that given a probability distribution $\rho_i(t)$ over states, the probability distribution at one infinitesimal time Δt later can be obtained from

$$\rho_i(t + \Delta t) = \rho_i(t) + \sum_j -k_{ji}\rho_i(t)\mathrm{d}t + k_{ij}\rho_j(t)\mathrm{d}t$$

where the important bit is that each $\rho(t)$ only depends on the state just before the current state. The matrix k_{ij} consists of constant real entries, which describe the rate at which the system can propagate from state j to state i. In other words the system is without memory which is usually known as the Markov approximation.

Kinetic Monte Carlo (kMC) integrates this equation by generating a state-to-state trajectory using a preset catalog of transitions or elementary steps and a rate constant for each elementary step. The reason to generate state-to-state trajectories rather than just propagating the entire probability distribution at once is that the transition matrix k_{ij} easily becomes too large for many systems at hand that even storing it would be too large for any storage device in foreseeable future.

As a quick estimate consider a system with 100 sites and 3 possible states for each site, thus having 3^{100} different configurations. The matrix to store all transition elements would have $(3^{100})^2 \approx 2.66\ 10^{95}$ entries, which exceeds the number of atoms on our planet by roughly 45 orders of magnitude. [1] And even though most of these elements would be zero since the number of accessible states is usually a lot smaller, there seems to be no simple way to transform to and solve this irreducible matrix in the general case.

Thus it is a lot more feasible to take one particular configuration and figure out the next process as well as the time it takes to get there and obtain ensemble averages from time averages taken over a sufficiently long trajectory. The basic steps can be described as follows

2.1.2 Basic Kinetic Carlo Algorithm

- **Fix rate constants** k_{ij} initial state x_i, and initial time t
- while $t < t_{\max}$ do

[1] Wolfram Alpha's estimate for number of atoms on earth.

1. draw random numbers $R_1, R_2 \in]0, 1]$
2. find l such that $\sum_{j=1}^{l} k_{ij} < k_{i,\mathrm{tot}} R_1 < \sum_{j_1}^{l+1} k_{ij}$
3. increment time $t \to t - \frac{\ln(R_2)}{k_{i,\mathrm{tot}}}$

end

2.1.3 Justification of the Algorithm

Let's understand why this simulates a physical process. The Markov approximation mentioned above implies several things: not only does it mean one can determine the next process from the current state. It also implies that all processes happen independently of one another because any memory of the system is erased after each step. Another great simplification is that rate constants simply add to a total rate, which is sometimes referred to as Matthiessen's rule, *viz* the rate with which *any* process occurs is simply $\sum_i k_i$.

First, one can show that the probability that n such processes occur in a time interval t is given by a Poisson distribution [2]

$$P(n, t) = \frac{\mathrm{e}^{-k_{\mathrm{tot}} t} (k_{\mathrm{tot}} t)^n}{n!}.$$

The waiting time or escape time t_w between two such processes is characterized by the probability that *zero* such processes have occured

$$P(0, t_w) = \mathrm{e}^{-k_{\mathrm{tot}} t_w}, \tag{2.1}$$

which, as expected, leads to an average waiting time of

$$\langle t_w \rangle = \frac{\int_0^\infty \mathrm{d}t_w \; t_w \mathrm{e}^{-k_{\mathrm{tot}} t_w}}{\int_0^\infty \mathrm{d}t_w \; \mathrm{e}^{-k_{\mathrm{tot}} t_w}} = \frac{1}{k_{\mathrm{tot}}}.$$

Therefore at every step, we need to advance the time by a random number that is distributed according to (2.1). One can obtain such a random number from a uniformly distributed random number $R_2 \in]0, 1]$ via $-\ln(R_2)/k_{\mathrm{tot}}$. [3]

Second, we need to select the next process. The next process occurs randomly but if we did this a very large number of times for the same initial state the number of times each process is chosen should be proportional to its rate constant. Experimentally one could achieve this by randomly sprinkling sand over an arrangement of buckets, where the size of the bucket is proportional to the rate constant and count each hit by a grain of sand in a bucket as one executed process. Computationally the same is achieved by steps 2 and 3.

2.1.4 Further Reading

For a very practical introduction I recommend Arthur Voter's tutorial [4] and Fichthorn [5] for a derivation, why Δt is chosen they way it is. The example given there is also an excellent exercise for any beginning kMC modeler. For recent review on implementation techniques I recommend the review by Reese et al. [6] and for a review over status and outlook I recommend the one by Reuter [7] .

[2] C. Gardiner, 2004. *Handbook of Stochastic Methods: for Physics, Chemistry, and the Natural Sciences*. Springer, 3rd edition, ISBN:3540208828.

[3] P. W. H, T. S. A, V. W. T, and F. B. P, 2007 *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Pres, 3rd edition, ISBN:0521768589, p. 287. link

[4] Voter, Arthur F. "Introduction to the Kinetic Monte Carlo Method." In Radiation Effects in Solids, 1–23, 2007. http://dx.doi.org/10.1007/978-1-4020-5295-8_1. link

[5] Fichthorn, Kristen A., and W. H. Weinberg. "Theoretical Foundations of Dynamical Monte Carlo Simulations." The Journal of Chemical Physics 95, no. 2 (July 15, 1991): 1090–1096. link

[6] Reese, J. S., S. Raimondeau, and D. G. Vlachos. "Monte Carlo Algorithms for Complex Surface Reaction Mechanisms: Efficiency and Accuracy." Journal of Computational Physics 173, no. 1 (October 10, 2001): 302–321. link

[7] Reuter, Karsten. "First-principles Kinetic Monte Carlo Simulations for Heterogeneous Catalysis: Concepts, Status and Frontiers". Wiley-VCH, 2009. link

2.2 Modelling Workflows

At the core of modelling lies the art to capture the most important features of a system and leave all others out. kmos is designed around the fact that modelling is a creative and iterative process.

A typical type of approach for modelling could be:

1. start with educated guess
2. calculate outcome
3. compare various observables and qualitative behavior with reference system
4. adapt model, goto 2. or publish model

So while this procedure is quite generic it may help to illustrate that the chances to find and capture the relevant features of a system are enhanced if the trial/learn loop is as short as possible.

2.2.1 kMC Modeling

A good way to define a model is to use a paper and pencil to draw your lattice, choose the species that you will need, draw each process and write down an expression for each rate constant, and finally fix all energy barriers and external parameters that you will need. Putting a model prepared like this into a computer is a simple exercise. You enter a new model by filling in

- meta information
- lattice
- species
- parameters
- processes

in roughly this order or open an existing one by opening a kMC XML file.

If you want to see the model run *kmos export <xml-file>* and you will get a subfolder with a self-contained Fortran90 code, which solves the model. If all necessary dependencies are installed you can simply run *kmos view* in the export folder.

2.2.2 kmos workflows

Since *kmos* has several entry points, there are several ways of using it. This section will outline different ways of using kmos:

- *the render script*

  Just write complete scripts as outlined in `../tutorials/first_model_api`. Export source from there or inspect XML file with one of the next methods below.

- *the GUI editor*

  Open an existing project *.xml file with

```
kmos edit <project_name>.xml
```

  and inspect or edit it through on screen

- *the CLI editor*

  Open an existing project *.xml file with

```
kmos import <project_name>.xml
```

  and edit the project interactively on the ipython console.

- *edit the XML file*

  Just open the XML file of your kmos project with a text editor of your choice and inspect or your model right there. This might only be a last resort to figure out what is going on. XML is often not considered very readable and note that changing variable names in one place might often break inconsistencies in other.

2.3 The kmos data model

The guide explains how kmos handles represent a kmc model internally, which is important to know if one wants to write new functionality.

The different functions and front-ends of kmos all interact in some way or another with instances of the Project class. A Project instance is a representation of a kmc model. If you fire up 'kmos edit' with an xml file, kmos validates the XML file and stores the content in a Project instance. If you export source code, kmos runs over the Project and creates the necessary Fortran 90 source code.

So the following things are in a Project:

- meta
- lattice(layers)
- species
- parameters
- processes

The language used here stems from modelling atomic movement on a fixed or evolving lattice like structure. In a more general context one may rephrase them as :

- meta -> information about project
- lattice -> geometry
- species -> states
- parameters
- processes -> transitions

2.4 How the kmos kMC algorithm works

kmos asks you to describe your model to the processor in seemingly arcane ways. It can save model descriptions in XML but they are basically unreadable and a pain to edit. The API has some glitches and is probably incomplete: so why learn it?

Because it is fast (in two ways).

The code it produces is commonly faster than naive implementations of the kMC method. Most straightforwards implementations of kMC take a time proportional to 2*N per kMC step, where N is the number of sites in the system. However the code that kmos produces is O(1) until the RAM of your system is exceeded. As benchmarks have shown this may happen when 100,000 or more sites are required. However tests have also shown that kmos can be faster than O(N) implementations from around 60-100 sites. If you have different experiences please let me know but I think this gives some rule of thumb.

Why is it faster? Straightforward implementations of kMC scan the entire system twice per kMC step. First to determine the total rate, then to determine the next process to be executed. The present implementation does not. kmos keeps a database of available processes which allow to quickly pick the next process. It also updates the database of available processes which cost additional overhead. However this overhead is independent of the system's size and only scales with the degree of interaction between sites, which is seems hard to define in general terms.

The second way reason why it is fast is because you can formulate processes in a intuitive fashion and let kmos figure how to make fast running code out of it. So we save in human time and CPU time, which is essentially human time as well. Yay!

To illustrate just how fast the algorithm is the graph below shows the CPU time needed to simulate 1 million kMC steps on a simple cubic lattice in 2 dimension with two reacting species and without lateral interaction. As this shows the CPU time spent per kMC step as nearly constant for up nearly 10^5 sites.

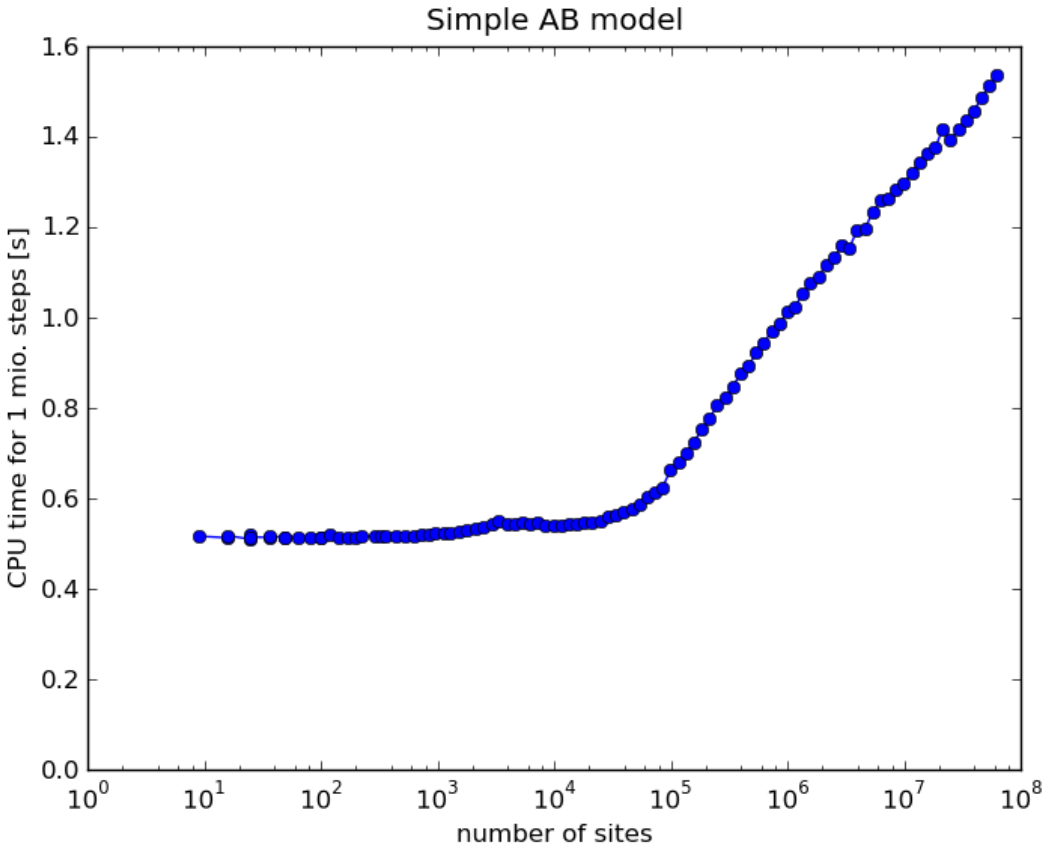

Fig. 2.1: Benchmark for a simple surface reaction model. All simulations have been performed on a single CPU of Intel I7-2600K with 3.40 GHz clock speed.

2.4.1 The kmos O(1) solver

So what makes the kMC solver so furiously fast? The underlying data structure is shown in the picture above. The most important part is that the solver never scans the entire system for available processes except at program initialization.

Please have a look at the sketch of data structures above. Given that all arrays are initialized and populated, in each kMC step the following things happen:

In the first step we need to identify the next process and site. To do so we draw a random number $R_1 \in [0, 1]$. This number has to be scaled to k_{tot}, so we multiply it with the last field in *accum. rates*. Next we simply perform a binary search for the right process on *accum. rates*. Having determined the process, we pick a site using a second random number R_2, which is constant in time since *avail sites* is filled up with the available site for each process from the left.

Totally independent of this we calculate the duration of the current step with another random number R_3 using

$$\Delta t = \frac{-\log(R_3)}{k_{\mathrm{tot}}}$$

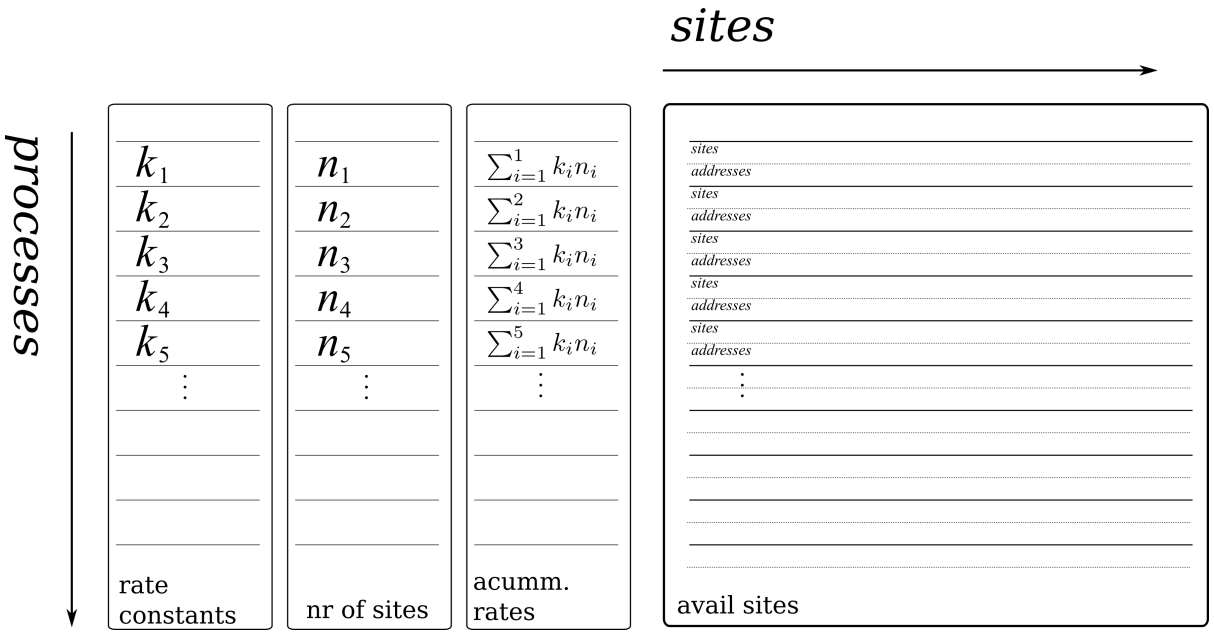

Fig. 2.2: The data model underlying the kmos solver. The central component is the *avail_sites* array which stores for each elementary step the sites for which it is executable. Secondly it stores the location in memory, where the availability of the site is stored for direct access. The array of *rate constants* holds the numeric rate constant and only changes, when a physical parameter is changed. The *nr of sites* array holds the total number of sites for each process and needs to be updated whenever a process becomes available und unavailable. The *accum. rates* has to be updated once per kMC step and holds the accumulated rate constant for each processes. That is, the last field of accum. rates holds k_{tot}, the total rate of the system.

So, while the determination of process and site is extremely straightforward, the CPU intensive part just starts now. The *proclist* module is written in such a way, for each elementary step it updates the *avail sites* array only in the local neighborhood of the site, where the process is executed. It is furthermore heuristically optimized in order to require only a minimal number of *if*-statement to figure out which database updates are necessary. This will be explained in greate detail in the next subsection.

For the current description it is sufficient to know that for all database updates by the *proclist* module :

- the *nr of sites* array is updated as well.
- adding or deleting an available site only takes constant time, since the number of available sites as well as the memory addresses is always updated. Thus new sites are simply add at the end of the list of available sites. When a site has to be deleted the last site in the array is moved to the memory slot available now.

Thus once all local updates are finished the *accum. rates* array is simply updated once. And ready we are for the next kMC step.

Todo

describe translation algorithm

2.5 The Process Syntax

In kMC language a process is uniquely defined by a configuration *before* the process is executed, a configuration *after* the process is executed, and a rate constant. Here this model is used to define a process by giving it a :

- condition_list
- action_list

- rate_constant

As you might guess each *condition* corresponds to one *before*, and each *action* coresponds to one *after*. In fact conditions and actions are actually of the same class or data type: each condition and action consists of a coordinate and a species which has to *be* or *will be* at the coordinate. This model of process definition also means that each process in one unit cell has to be defined explicitly. Typically one a single crystal surface one will have not one diffusion per species but as many as there are equivalent directions :

- species_diffusion_right
- species_diffusion_up
- species_diffusion_left
- species_diffusion_down

while this seems like a lot of work to define that many processes, it allows for a very clean and simple definition of a process itself. Later you can use geometric measures to abstract these cases as you will see further down.

2.5.1 Adsorption

Let's start with a very simple and basic process: molecular adsorption of a gas phase species, let call it `A` on a surface site. For this we need a species

```
from kmos.types import *
pt = Project()

A = Species(name='A')
pt.add_species(A)

empty = Species(name='empty')
pt.add_species(empty)
```

and the coordinate of a surface site

```
layer = Layer(name='default')
pt.add_layer(layer)
layer.sites.append(Site(name='a'))
coord = pt.lattice.generate_coord('a.(0,0,0).default')
```

which is for now all we need to define an adsorption process:

```
adsorption = Proces(name='adsorption_A_a',
                    condition_list=[Condition(coord=coord,
                                             species='empty')],
                    action_list=[Action(coord=coord,
                                        species='A')])
pt.add_process(adsorption)
```

Now this wasn't hard, was it?

2.5.2 Diffusion

Let's move to another example, namely the *diffusion* of a particle to the next unit cell in the y-direction. You first need the coordinate of the final site

```
final = pt.lattice.generate_coord('a.(0,1,0).default')
```

and you are good to go

```
diffusion_up = Process('diffusion_A_up',
                       condition_list=[Condition(coord=coord,
                                                species='A'),
```

```
                                        Condition(coord=final,
                                                  species='empty')],
                          condition_list=[Condition(coord=coord,
                                                  species='empty'),
                                        Condition(coord=final,
                                                  species='A')],
pt.add_process(diffusion_up)
```

You can complicated this *ad infinitum* but you know all elements needed to define processes.

2.5.3 Avoid Double Counting

Finally a word of warning: *double counting* is a phenomenon sometimes encountered for those process where there is more than one equivalent direction for a process and the coordinates within the process are also equivalent. Think of dissociative oxygen adsorption. Novices typically collect all possible directions (e.g. right, up, left, down) and then define this process for each direction. Later they realize that in fact they *double counted* the process because e.g. adsorption_up is the same processes as adsorption_down, just executed from one site above or below. Then they compensate by dividing each adsorption rate constant by 2. Later realizing that they have to do the same for desorption. Ok, I have done this and believe me it is really bad when you are looking for an error if at the same you already divide the unit cell size by 2 for some reason.

The smart way out is to save the pain and to avoid double counting completely from the beginning and just think how many process are geometrically inequivalent in the unit cell. A simple trick is to only consider processes in the *positive* directions.

2.5.4 Taking It Home

- A process consists of conditions, actions and a rate constant
- *double counting* is best avoided from the beginning

2.6 The Site/Coordinate Syntax

In the atomistic kMC simulations pursued here one defines processes in terms of sites on some more or less fixed lattice. This reflects the physical observation that molecules on surfaces adsorb on very specific locations above a solid.

To represent this in a computer program, we first need to make a small but crucial differentiation: namely the difference between the *sites* of a (surface) structure and the *coordinates* of a process. The difference is that a given structure contains each site defined exactly once, whereas a process may use the same site several times however in a different unit cell. So this differentation owes to the fact that we commonly simulate highly periodic structures.

Ok, having this out of the way you start to define and use sites and coordinates. The minimum constructor for a site is

```
site = Site(name='site_name')
```

where `site_name` can be a string without spaces and all names should be unique within one layer. Usually it is reasonable to add a position in relative coordinates right-away like so

```
site = Site(name='hollow', pos='0.5 0.5 0.0')
```

which would place the site at the bottom center of the cell. A direct benefit is that you can measure distances between coordinates later on to, e.g. select all nearest neighbor or next-nearest neighbor sites.

A site can have some more attributes. Some of them are only needed in conjunction with GUI use. It is worth to know that each site can have one or more tags. This way one create types of site and conveniently select all sites with a one more tags. The syntax here is as follows

```
site = Site(name='hollow', pos='0.5 0.5 0.0', tags='tag1 tag2 ...')
```

The second part is to generate the coordinates that are used in the process description.

2.6.1 Manual generation

To quickly generate single coordinates you can generate it from a Project like so

```
pt.lattice.generate_coord('hollow.(0,0,0).layer_name')
```

Let's look at the generation string. The general syntax is

```
site_name.offset.layer_name
```

The `site_name` and the `layer_name` must have been defined before. The offset is a tuple of three integer numbers *(0, 0, 1)* and specifies the relative unit cell of this coordinate. Of course this only becomes meaningful as soon as you use more than one coordinate in a process.

Missing values will be filled in from the back using default values, such that

```
site -> site.(0,0,0) -> site.(0,0,0).default_layer
```

2.6.2 Advanced Coordinate Techniques

Generating large process lists with a lot of similar or even degenerate processes is a very boring task. So we should try to use programming logic as much as possible. Here I will outline a couple of idioms you can use here.

Often times it is handier (less typing) to generate a larger set of coordinates at first and then select different subsets from it in a process definition. For this purpose you can use

```
pset = pt.lattice.generate_coord_set(size=[x,y,z], layer_name='layer_name')
```

This collects all sites from the given layer and generates all coordinates in the first unit cell (`offset=(1,1,1)`) and all `x`, `y`, and `z` unit cells in the respective direction.

To select subsets in a readable way I suggest you use list comprehensions, like so

```
[ x for x in pset if not x.offset.any() ]
```

which again selects all sites in the first unit cell. Or to select all site tagged with `foo` you could use

```
[ x for x in pset if 'foo' in x.tags.split() ]
```

or having defined a unit cell size and a site position your can measure real-space distances between coordinate like so

```
np.linalg.norm(x.pos-y.pos)
```

Or of course you can use any combination of the above.

2.6.3 Taking it home

- *sites* belong to a *structure* while *coordinates* belong to a *process*
- coordinates are generated from sites
- coordinate sets can be selected and chopped using list comprehensions and tags

CHAPTER 3

Reference

3.1 Command Line Interface (CLI)

Entry point module for the command-line interface. The kmos executable should be on the program path, import this modules main function and run it.

To call kmos command as you would from the shell, use

```
kmos.cli.main('...')
```

Every command can be shortened as long as it is non-ambiguous, e.g.

```
kmos ex <xml-file>
```

instead of

```
kmos export <xml-file>
```

etc.

3.1.1 List of commands

`kmos benchmark` Run 1 mio. kMC steps on model in current directory and report runtime.

`kmos build` Build kmc_model.so from *f90 files in the current directory.

Additional Parameters ::

-d/–debug Turn on assertion statements in F90 code

-n/–no-compiler-optimization Do not send optimizing flags to compiler.

`kmos edit <xml-file>` Open the kmos xml-file in a GUI to edit the model.

`kmos export <xml-file> [<export-path>]` Take a kmos xml-file and export all generated source code to the export-path. There try to build the kmc_model.so.

Additional Parameters

```
-s/--source-only
    Export source only and don't build binary

-b/--backend (local_smart|lat_int)
    Choose backend. Default is "local_smart".
    lat_int is EXPERIMENTAL and not made
    for production, yet.

-d/--debug
    Turn on assertion statements in F90 code.
    (Only active in compile step)
```

```
    -n/--no-compiler-optimization
        Do not send optimizing flags to compiler.
```

kmos help <command> Print usage information for the given command.

kmos help all Display documentation for all commands.

kmos import <xml-file> Take a kmos xml-file and open an ipython shell with the project_tree imported as pt.

kmos rebuild Export code and rebuild binary module from XML information included in kmc_settings.py in current directory.

Additional Parameters ::

-d/–debug Turn on assertion statements in F90 code

kmos run

Open an interactive shell and create a KMC_Model in it run == shell

kmos settings-export <xml-file> [<export-path>] Take a kmos xml-file and export kmc_settings.py to the export-path.

kmos shell

Open an interactive shell and create a KMC_Model in it run == shell

kmos version Print version number and exit.

kmos view Take a kmc_model.so and kmc_settings.py in the same directory and start to simulate the model visually.

Additional Parameters ::

-v/–steps-per-frame <number> Number of steps per frame

kmos xml Print xml representation of model to stdout

3.2 Data Types

3.2.1 kmos.types

Holds all the data models used in kmos.

class kmos.types.**Project**

A Project is where (almost) everything comes together. A Project holds all other elements needed to describe one kMC Project ready to be manipulated, exported, or imported.

The overall structure is the following as is also displayed in the editor GUI.

Project:

```
- Meta
- Parameters
- Lattice(s)
- Species
- Processes
```

add_layer(**layers*, ***kwargs*)

Add a layer to the project. A Layer, or keywords that are passed to the Layer constructor are accepted.

Parameters

- **layers** (*list*) – List of layers.
- **cell** (*np.array (3x3)*) – Size of unit-cell.

- **default_layer** (*str.*) – name of default layer.

add_parameter (**parameters*, ***kwargs*)
Add a parameter to the project. A Parameter, or keywords that are passed to the Parameter constructor are accepted.

Parameters

- **name** (*str*) – The name of the parameter.
- **value** (*float*) – Default value of parameter.
- **adjustable** (*bool*) – Create controller in GUI.
- **min** (*float*) – Minimum value for controller.
- **max** (*float*) – Maximum value for controller.
- **scale** (*str*) – Controller scale: 'log' or 'lin'

add_process (**processes*, ***kwargs*)
Add a process to the project. A Process, or keywords that are passed to the Process constructor are accepted.

Parameters

- **name** (*str*) – Name of process.
- **rate_constant** (*str*) – Expression for rate constant.
- **condition_list** (*list.*) – List of conditions (class Condition).
- **action_list** (*list.*) – List of conditions (class Action).
- **enabled** (*bool.*) – Switch this process on or of.
- **chemical_expression** (*str.*) – Chemical expression (i.e: A@site1 + B@site2 -> empty@site1 + AB@site2) to generate process from.
- **tof_count** (*dict.*) – Stoichiometric factor for observable products {'NH3': 1, 'H2O(gas)': 2}. Hint: avoid space in keys.

add_site (***kwargs*)
Add a site to the project. The arguments are

add_site(layer_name, site)

Parameters

- **name** (*str*) – Name of layer to add the site to.
- **site** (Site) – Site instance to add.

add_species (**speciess*, ***kwargs*)
Add a species to the project. A Species, or keywords that are passed to the Species constructor are accepted.

Parameters

- **name** (*str*) – Name of species.
- **color** (*str*) – Color of species in editor GUI (#ffffff hex-type specification).
- **representation** (*str*) – ase.atoms.Atoms constructor describing species geometry.
- **tags** (*str*) – Tags of species (space separated string).

get_parameters (*pattern=None*)
Return list of parameters in Project.

Parameters **pattern** (*str*) – Pattern to fnmatch name of parameter against.

get_processes(*pattern=None*)
Return list of processes.

Parameters **pattern** (*str*) – Pattern to fnmatch name of process against.

get_speciess(*pattern=None*)
Return list of species in Project.

Parameters **pattern** (*str*) – Pattern to fnmatch name of process against.

import_xml_file(*filename*)
Takes a filename, validates the content against kmc_project.dtd and import all fields into the current project tree

parse_and_add_process(*string*)
Generate and add processes using a shorthand notation like, e.g. :: process_name; species1A@coord1 + species2A@coord2 + ... -> species1B@coord1 + species2A@coord2 + ...; rate_constant_expression

.

Parameters **string** (*str*) – shorthand notation for process

parse_process(*string*)
Generate processes using a shorthand notation like, e.g. :: process_name; species1A@coord1 + species2A@coord2 + ... -> species1B@coord1 + species2A@coord2 + ...; rate_constant_expression

.

Parameters **string** (*str*) – shorthand notation for process

validate_model()
Run various consistency and completeness test of the model to make sure we have a minimally complete model.

class kmos.types.**Meta**(**args*, ***kwargs*)
Class holding the meta-information about the kMC project

class kmos.types.**Parameter**(***kwargs*)
A parameter that can be used in a rate constant expression and defined via some init file.

Parameters

- **name** (*str*) – The name of the parameter.
- **adjustable** (*bool*) – Create controller in GUI.
- **min** (*float*) – Minimum value for controller.
- **max** (*float*) – Maximum value for controller.
- **scale** (*str*) – Controller scale: 'log' or 'lin'

class kmos.types.**LayerList**(***kwargs*)
A list of layers

Parameters

- **cell** (*np.array (3x3)*) – Size of unit-cell.
- **default_layer** (*str.*) – name of default layer.

generate_coord(*terms*)
Expecting something of the form site_name.offset.layer and return a Coord object

generate_coord_set(*size=[1, 1, 1], layer_name='default'*)
Generates a set of coordinates around unit cell of any desired size. By default it includes exactly all sites in the unit cell. By setting size=[2,1,1] one gets an additional set in the positive and negative x-direction.

class kmos.types.**Layer**(***kwargs*)
Represents one layer in a possibly multi-layer geometry.

Parameters

- **name** (*str*) – Name of layer.
- **sites** (*list*) – Sites associated with this layer (Default: [])

class kmos.types.**Site**(***kwargs*)
Represents one lattice site.

Parameters

- **name** (*str*) – Name of site.
- **pos** (*np.array or str*) – Position within unit cell.
- **tags** (*str*) – Tags for this site (space separated).
- **default_species** (*str*) – Initial population for this site.

class kmos.types.**Species**(***kwargs*)
Class that represent a species such as oxygen, empty, ... Note: *empty* is treated just like a species.

Parameters

- **name** (*str*) – Name of species.
- **color** (*str*) – Color of species in editor GUI (#ffffff hex-type specification).
- **representation** (*str*) – ase.atoms.Atoms constructor describing species geometry.
- **tags** (*str*) – Tags of species (space separated string).

class kmos.types.**Process**(***kwargs*)
One process in a kMC process list

Parameters

- **name** (*str*) – Name of process.
- **rate_constant** (*str*) – Expression for rate constant.
- **condition_list** (*list.*) – List of conditions (class Condition).
- **action_list** (*list.*) – List of conditions (class Action).
- **enabled** (*bool.*) – Switch this process on or of.
- **chemical_expression** (*str.*) – Chemical expression (i.e: A@site1 + B@site2 -> empty@site1 + AB@site2) to generate process from.
- **tof_count** (*dict.*) – Stoichiometric factor for observable products {'NH3': 1, 'H2O(gas)': 2}. Hint: avoid space in keys.

class kmos.types.**ConditionAction**(***kwargs*)
Represents either a condition or an action. Since both have the same attributes we use the same class here, and just store them in different lists, depending on its role. For better readability one can also use *Condition* or *Action* which are just aliases.

Parameters

- **coord** (Coord) – Relative Coord (generated by *LayerList.generate_coord()* or Lattice.generate_coord_set()).
- **species** (*str*) – Name of species.

class kmos.types.**Coord**(***kwargs*)
Class that holds exactly one coordinate as used in the description of a process. The distinction between a Coord and a Site may seem superfluous but it is made to avoid data duplication.

Parameters

- **name** (*str*) – Name of coordinate.

- **offset** (*np.array or list*) – Offset in term of unit-cells.
- **layer** (*str*) – Name of layer.
- **tags** (*str*) – List of tags (space separated string).

pos
pos is np.array((3, 1)) and is calculated from offset and position. Not to be set manually.

3.2.2 kmos.io

Features front-end import/export functions for kMC Projects. Currently import and export is supported to XML and export is supported to Fortran 90 source code.

kmos.io.**export_source**(*project_tree*, *export_dir=None*, *code_generator='local_smart'*)
Export a kmos project into Fortran 90 code that can be readily compiled using f2py. The model contained in project_tree will be stored under the directory export_dir. export_dir will be created if it does not exist. The XML representation of the model will be included in the kmc_settings.py module.

export_source is *the* central feature of the *kmos* approach. In order to generate different *backend* solvers, additional candidates of this methods could be implemented.

kmos.io.**export_xml**(*project_tree*, *filename=None*)
Writes a project to an XML file.

class kmos.io.**ProcListWriter**(*data*, *dir*)
Write the different parts of Fortran 90 code needed to run a kMC model.

write_lattice()
Write the lattice.f90 module, i.e. the geometric information that belongs to a kMC model.

write_proclist(*smart=True*, *code_generator='local_smart'*)
Write the proclist.f90 module, i.e. the rules which make up the kMC process list.

write_settings()
Write the kmc_settings.py. This contains all parameters, which can be changed on the fly and without recompilation of the Fortran 90 modules.

3.3 Editor frontend

3.3.1 kmos.gui

A GUI frontend to create and edit kMC models.

class kmos.gui.**Editor**
The editor GUI frontend.

class kmos.gui.**GTKProject**(*parent*, *menubar*)
A facade of kmos.types.Project so that pygtk can display in a TreeView.

3.3.2 kmos.forms

3.4 Runtime frontend

3.4.1 kmos.run

3.4.2 kmos.view

3.4.3 kmos.cli

Entry point module for the command-line interface. The kmos executable should be on the program path, import this modules main function and run it.

To call kmos command as you would from the shell, use

```
kmos.cli.main('...')
```

Every command can be shortened as long as it is non-ambiguous, e.g.

```
kmos ex <xml-file>
```

instead of

```
kmos export <xml-file>
```

etc.

kmos.cli.**main**(*args=None*)
The CLI main entry point function.

The optional argument args, can be used to directly supply command line argument like

$ kmos <args>

otherwise args will be taken from STDIN.

kmos.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.

kmos.cli.**sh**(*banner*)
Wrapper around interactive ipython shell that factors out ipython version depencies.

3.5 Utils

3.5.1 kmos.utils

Several utility functions that do not seem to fit somewhere else.

class kmos.utils.**CorrectlyNamed**
Syntactic Sugar class for use with kiwi, that makes sure that the name field of the class has a name field, that always complys with the rules for variables.

__init__()

__module__ = 'kmos.utils'

on_name__validate(*_*, *name*)
Called by kiwi upon chaning a string

kmos.utils.**T_grid**(*T_min*, *T_max*, *n*)

kmos.utils.**build**(*options*)
Build binary with f2py binding from complete set of source file in the current directory.

kmos.utils.**col_str2tuple**(*hex_string*)
Convenience function that turns a HTML type color into a tuple of three float between 0 and 1

kmos.utils.**col_tuple2str**(*tup*)
Convenience function that turns a HTML type color into a tuple of three float between 0 and 1

kmos.utils.**download**(*project*)

kmos.utils.**evaluate_kind_values**(*infile*, *outfile*)
Go through a given file and dynamically replace all selected_int/real_kind calls with the dynamically evaluated fortran code using only code that the function itself contains.

kmos.utils.**evaluate_param_expression**(*param*, *parameters={}*)

kmos.utils.**evaluate_rate_expression**(*rate_expr*, *parameters={}*)
Evaluates an expression for a typical kMC rate constant. This expression can be any python expression returning a number (most likely you will want a float). Standard functions like sin(x), cos(x), pi, pow(x,y), log(x, [base]) are evaluated using the builtin math module.

beta is an shorthand for 1/(kboltzmann*T) and requires the present of a temperature parameter *T*.

Short-hands for common conversion factor (bar, c, hbar, h, hbar, e, eV, angstrom, umass) are evaluated as defined in *kmos.units*.

The short-hand *m_CO* is substituted by the atomic weight in atomic mass units *u*.

The short-hand *p_<species>* is interpreted as the gas-phase pressure of the corresponding species which can be used in other functions.

The short-hand *mu_<species>* is interpreted as the gas-phase chemical potential in eV which kmos attempts to evaluate using a linear interpolation of the corresponding JANAF gas phase chemical table (if installed).

Additional external parameters can be passed in as dictionary, like the following:

parameters = {'p_CO':{'value':1}, 'T':{'value':1}}

or as a list of parameters:

parameters = [Parameter(), ...]

kmos.utils.**get_ase_constructor**(*atoms*)
Return the ASE constructor string for *atoms*.

kmos.utils.**jmolcolor_in_hex**(*i*)
Return a given jmol color in hexadecimal representation.

kmos.utils.**p_grid**(*p_min*, *p_max*, *n*)

kmos.utils.**product**(**args*, ***kwds*)
Take two lists and return iterator producing all combinations of tuples between elements of the two lists.

kmos.utils.**split_sequence**(*seq*, *size*)
Take a list and a number n and return list divided into n sublists of roughly equal size.

kmos.utils.**timeit**(*func*)
Generic timing decorator

To stop time for function call f just

```
from kmos.utils import timeit
@timeit
def f():
    ...
```

kmos.utils.**write_py**(*fileobj*, *images*, ***kwargs*)
Write a ASE atoms construction string for *images* into *fileobj*.

3.6 kmos kMC project DTD

The central storage and exchange format is XML. XML was chosen over JSON, pickle or alike because it still seems as the most flexible and universal format with good methods to define the overall structure of the data.

One way to define an XML format is by using a document type description (DTD) and in fact at every import a kmos file is validated against the DTD below.

```
<!ELEMENT kmc (meta?,species_list?,parameter_list?, lattice, process_list?,output_list?)>
  <!ATTLIST kmc
  version CDATA #REQUIRED
  >
  <!ELEMENT meta EMPTY>
  <!ATTLIST meta
    author CDATA #IMPLIED
    debug CDATA #IMPLIED
    email CDATA #IMPLIED
    model_dimension CDATA #IMPLIED
    model_name CDATA #IMPLIED
  >

  <!ELEMENT species_list (species)*>
  <!ATTLIST species_list
    default_species CDATA #IMPLIED
  >
  <!ELEMENT species EMPTY>
  <!ATTLIST species
    name CDATA #REQUIRED
    color CDATA #IMPLIED
    representation CDATA #IMPLIED
    tags CDATA #IMPLIED
  >
  <!ELEMENT parameter_list (parameter)*>
  <!ELEMENT parameter EMPTY>
  <!ATTLIST parameter
    name CDATA #REQUIRED
    value CDATA #IMPLIED
    adjustable CDATA #IMPLIED
    min CDATA #IMPLIED
    max CDATA #IMPLIED
    scale CDATA #IMPLIED
  >
  <!ELEMENT lattice (layer)*>
  <!ATTLIST lattice
    cell_size CDATA #REQUIRED
    default_layer CDATA #REQUIRED
    substrate_layer CDATA #IMPLIED
    representation CDATA #IMPLIED
  >
  <!ELEMENT layer (site)*>
  <!ATTLIST layer
    name CDATA #REQUIRED
    grid CDATA #IMPLIED
    grid_offset CDATA #IMPLIED
    color CDATA #IMPLIED
  >
  <!ELEMENT site EMPTY>
  <!ATTLIST site
    pos CDATA #REQUIRED
    type CDATA #REQUIRED
    tags CDATA #IMPLIED
    default_species CDATA #IMPLIED
  >
```

```
  <!ELEMENT process_list (process)*>
    <!ELEMENT process (condition|action)*>
      <!ATTLIST process
        name CDATA #REQUIRED
        rate_constant CDATA #REQUIRED
        enabled CDATA #IMPLIED
        tof_count CDATA #IMPLIED
      >
      <!ELEMENT condition EMPTY>
      <!ATTLIST condition
        coord_name CDATA #REQUIRED
        coord_layer CDATA #REQUIRED
        coord_offset CDATA #REQUIRED
        species CDATA #REQUIRED
        implicit CDATA #IMPLIED
      >
      <!ELEMENT action EMPTY>
      <!ATTLIST action
        coord_name CDATA #REQUIRED
        coord_layer CDATA #REQUIRED
        coord_offset CDATA #REQUIRED
        species CDATA #REQUIRED
      >
      <!ELEMENT output_list (output)*>
        <!ELEMENT output EMPTY>
        <!ATTLIST output
          item CDATA #REQUIRED
        >
```

3.7 Backend

In general the backend includes all functions that are implemented in Fortran90, which therefore should not have to be changed by hand often. The backend is divided into three modules, which import each other in the following way

```
base <- lattice <- proclist
```

The key for this division is reusability of the code. The *base* module implement all aspects of the kMC code, which do not depend on the described model. Thus it "never" has to change. The *latttice* module basically repeats all methods of the *base* model in terms of lattice coordinates. Thus the *lattice* module only changes, when the geometry of the model changes, *e.g.* when you add or delete sites. The *proclist* module implements the process list, that is the species or states each site can have and the elementary steps. Typically that changes most often while developing a model.

The rate constants and physical parameters of the system are not implemented in the backend at all, since in the physical sense they are too high-level to justify encoding and compilation at the Fortran level and so they are typical read and parsed from a python script.

The *kmos.run.KMC_Model* class implements a convenient interface for most of these functions, however all public methods (in Fortran called subroutines) and variables can also be accessed directly like so

```
from kmos.run import KMC_Model
model = KMC_Model(print_rates=False, banner=False)
model.base.<TAB>
model.lattice.<TAB>
model.proclist.<TAB>
```

which works best in conjunction with ipython.

3.7.1 kmos/base

The base kMC module, which implements the kMC method on a $d = 1$ lattice. Virtually any lattice kMC model can be build on top of this. The methods offered are:

- de/allocation of memory
- book-keeping of the lattice configuration and all available processes
- updating and tracking kMC time, kMC step and wall time
- saving and reloading the current state
- determine the process and site to be executed

base/accum_rates

Stores the accumulated rate constant multiplied with the number of sites available for that process to be used by determine_procsite. Let $\mathbf{c}$ be the rate constants $\mathbf{n}$ the number of available sites, and $\mathbf{a}$ the accumulated rates, then a_i is calculated according to $a_i = \sum_{j=1}^{i} c_j n_j$.

base/add_proc

The main idea of this subroutine is described in del_proc. Adding one process to one capability is programmatically simpler since we can just add it to the end of the respective array in avail_sites.

- `proc` positive integer number that represents the process to be added.
- `site` positive integer number that represents the site to be manipulated

base/allocate_system

Allocates all book-keeping structures and stores local copies of system name and size(s):

- `systen_name` identifier of this simulation, used as name of punch file
- `volume` the total number of sites
- `nr_of_proc` the total number of processes

base/assertion_fail

Function that shall be used by all parts of the program to print a proper message in case some assertion fails.

- `a` condition that is supposed to hold true
- `r` message that is printed to the poor user in case it fails

base/avail_sites

Main book-keeping array that stores for each process the sites that are available and for each site the address in this very array. The meaning of the fields are:

avail_sites(proc, field, switch)

where:

- proc – refers to a process in the process list
- the field within the process, but the meaning differs as explained under 'switch'

- switch – can be either 1 or 2 and switches between (1) the actual numbers of the sites, which are available and filled in from the left but in whatever order they come or (2) the location where the site is stored in (1).

base/can_do

Returns true if 'site' can do 'proc' right now

- `proc` integer representing the requested process.
- `site` integer representing the requested site.
- `can` writeable boolean, where the result will be stored.

base/deallocate_system

Deallocate all allocatable arrays: avail_sites, lattice, rates, accum_rates, integ_rates, procstat.

`none`

base/del_proc

del_proc delete one process from the main book-keeping array avail_sites. These book-keeping operations happen in O(1) time with the help of some more book-keeping overhead. avail_sites stores for each process all sites that are available. The array for each process is filled from the left, but sites generally not ordered. With this determine_procsite can effectively pick the next site and process. On the other hand a second array (avail_sites(:,:,2)) holds for each process and each site, the location where it is stored in avail_site(:,:,1). If a site needs to be removed this subroutine first looks up the location via avail_sites(:,:,1) and replaces it with the site that is stored as the last element for this process.

- `proc` positive integer that states the process
- `site` positive integer that encodes the site to be manipulated

base/determine_procsite

Expects two random numbers between 0 and 1 and determines the corresponding process and site from accum_rates and avail_sites. Technically one random number would be sufficient but to circumvent issues with wrong interval_search_real implementation or rounding errors I decided to take two random numbers:

- `ran_proc` Random real number from $\in [0, 1]$ that selects the next process
- `ran_site` Random real number from $\in [0, 1]$ that selects the next site
- `proc` Return integer $\in [1, \text{nr\_of\_proc}$
- `site` Return integer $\in [1, \text{volume}$

base/get_accum_rate

Return accumulated rate at a given process.

- `proc_nr` integer representing the requested process.
- `return_accum_rate` writeable real, where the requested accumulated rate will be stored.

base/get_avail_site

Return field from the avail_sites database

- `proc_nr` integer representing the requested process.
- `field` integer for the site at question
- `switch` 1 or 2 for site or storage location

base/get_integ_rate

Return integrated rate at a given process.

- `proc_nr` integer representing the requested process.
- `return_integ_rate` writeable real, where the requested integrated rate will be stored.

base/get_kmc_step

Return the current kmc_step

- `kmc_step` Writeable integer

base/get_kmc_time

Returns current kmc_time as rdouble real as defined in kind_values.f90.

- `return_kmc_time` writeable real, where the kmc_time will be stored.

base/get_kmc_time_step

Returns current kmc_time_step (the time increment).

- `return_kmc_step` writeable real, where the kmc_time_step will be stored.

base/get_kmc_volume

Return the total number of sites.

- `volume` Writeable integer.

base/get_nrofsites

Return how many sites are available for a certain process. Usually used for debugging

- `proc` integer representing the requested process
- `return_nrofsites` writeable integer, where nr of sites gets stored

base/get_procstat

Return process counter for process proc as integer.

- `proc` integer representing the requested process.
- `return_procstat` writeable integer, where the process counter will be stored.

base/get_rate

Return rate of given process.

- `proc_nr` integer representing the requested process.
- `return_rate` writeable real, where the requested rate will be stored.

base/get_species

Return the species that occupies site.

- `site` integer representing the site

base/get_system_name

Return the systems name, that was specified with base/allocate_system

- `system_name` Writeable string of type character(len=200).

base/get_walltime

Return the current walltime.

- `return_walltime` writeable real where the walltime will be stored.

base/increment_procstat

Increment the process counter for process proc by one.

- `proc` integer representing the process to be increment.

base/integ_rates

Stores the time-integrated rates (non-normalized to surface area) Used to determine reaction rates, i.e. average number of reactions per unit surface and time. Let $\mathbf{a}$ the integrated rates, $\mathbf{c}$ be the rate constants, $\mathbf{n}_i$ the number of available sites during kMC-time interval i, $\{\Delta t_i\}$ the corresponding timesteps then $a_i(t)$ at the time $t = \sum_{i=1} \Delta t_i$ is calculated according to $a_i(t) = \sum_{i=1} c_i n_i \Delta t_i$.

base/interval_search_real

This is basically a standard binary search algorithm that expects an array of ascending real numbers and a scalar real and return the key of the corresponding field, with the following modification :

- the value of the returned field is equal of larger of the given value. This is important because the given value is between 0 and the largest value in the array and otherwise the last field is never selected.
- if two or more values in the array are identical, the function return the index of the leftmost of those field. This is important because having field with identical values means that all field except the leftmost one do not contain any sites. Refer to update_accum_rate to understand why.
- the value of the returned field may no be zero. Therefore the index the to be equal or larger than the first non-zero field.

However: as everyone knows the binary search is trickier than it appears at first site especially real numbers. So intensive testing is suggested here!

- `arr` real array of type rsingle (kind_values.f90) in monotonically (not strictly) increasing order

- `value` real positive number from [0, max_arr_value]

base/kmc_step

Number of kMC steps executed.

base/kmc_time

Simulated kMC time in this run in seconds.

base/kmc_time_step

The time increment of the current kMC step.

base/lattice

Stores the actual physical lattice in a 1d array, where the value on each slot represents the species on that site.

Species constants can be conveniently defined in lattice_... and later used directly in the process list.

base/nr_of_proc

Total number of available processes.

base/nr_of_sites

Stores the number of sites available for each process.

base/procstat

Stores the total number of times each process has been executed during one simulation.

base/rates

Stores the rate constants for each process in s^-1.

base/reload_system

Restore state of simulation from *.reload file as saved by save_system(). This function also allocates the system's memory so calling allocate_system again, will cause a runtime failure.

- `system_name` string of 200 characters which will make the reload_system look for a file called ./<system_name>.reload
- `reloaded` logical return variable, that is .true. reload of system could be completed successfully, and .false. otherwise.

base/replace_species

Replaces the species at a given site with new_species, given that old_species is correct, i.e. identical to the site that is already there.

- `site` integer representing the site
- `old_species` integer representing the species to be removed
- `new_species` integer representing the species to be placed

base/reset_site

This function is a higher-level function to reset a site as if it never existed. To achieve this the species is set to null_species and all available processes are stripped from the site via del_proc.

- `site` integer representing the requested site.
- `species` integer representing the species that ought to be at the site, for consistency checks

base/save_system

save_system stores the entire system information in a simple ASCII filed names <system_name>.reload. All fields except avail_sites are stored in the simple scheme:

variable value

In the case of array variables, multiple values are seperated by one or more spaces, and the record is terminated with a newline. The variable avail_sites is treated slightly differently, since printed on a single line it is almost impossible to interpret from the ASCII files. Instead each process starts a new line, and the first number on the line stands for the process number and the remaining fields, hold the values.

`none`

base/set_kmc_time

Sets current kmc_time as rdouble real as defined in kind_values.f90.

- `new` readable real, that the kmc time will be set to

base/set_rate_const

Allows to set the rate constant of the process with the number proc_nr.

- `proc_n` The process number as defined in the corresponding proclist_ module.
- `rate` the rate in s^{-1}

base/set_system_name

Set the systems name. Useful in conjunction with base.save_system to save *.reload files under a different name than the default one.

- `system_name` Readable string of type character(len=200).

base/start_time

CPU time spent in simulation at least reload.

base/system_name

Unique indentifier of this simulation to be used for restart files. This name should not contain any characters that you don't want to have in a filename either, i.e. only [A-Za-z0-9_-].

base/update_accum_rate

Updates the vector of accum_rates.

`none`

base/update_clocks

Updates walltime, kmc_step and kmc_time.

- `ran_time` Random real number $\in [0, 1]$

base/update_integ_rate

Updates the vector of integ_rates.

`none`

base/volume

Total number of sites.

base/walltime

Total CPU time spent on this simulation.

3.7.2 kmos/lattice

Implements the mappings between the real space lattice and the 1-D lattice, which kmos/base operates on. Furthermore replicates all geometry specific functions of kmos/base in terms of lattice coordinates. Using this module each site can be addressed with 4-tuple `(i, j, k, n)` where `i, j, k` define the unit cell and `n` the site within the unit cell.

lattice/allocate_system

Allocates system, fills mapping cache, and checks whether mapping is consistent

`none`

lattice/calculate_lattice2nr

Maps all lattice coordinates onto a continuous set of integer $\in [1, volume]$

- `site` integer array of size (4) a lattice coordinate

lattice/calculate_nr2lattice

Maps a continuous set of of integers $\in [1, volume]$ to a 4-tuple representing a lattice coordinate

- `nr` integer representing the site index

lattice/deallocate_system

Deallocates system including mapping cache.

`none`

3.7.3 kmos/proclist

Implements the kMC process list.

proclist/do_kmc_step

Performs exactly one kMC step. * first update clock * then configuration sampling step * last execute process

`none`

proclist/do_kmc_steps

Performs `n` kMC step. If one has to run many steps without evaluation do_kmc_steps might perform a little better. * first update clock * then configuration sampling step * last execute process

`n` : Number of steps to run

proclist/get_kmc_step

Determines next step without executing it.

`none`

proclist/get_occupation

Evaluate current lattice configuration and returns the normalized occupation as matrix. Different species run along the first axis and different sites run along the second.

`none`

proclist/init

Allocates the system and initializes all sites in the given layer.

- `input_system_size` number of unit cell per axis.
- `system_name` identifier for reload file.
- `layer` initial layer.
- `no_banner` [optional] if True no copyright is issued.

proclist/initialize_state

Initialize all sites and book-keeping array for the given layer.

- `layer` integer representing layer

proclist/run_proc_nr

Runs process `proc` on site `nr_site`.

- `proc` integer representing the process number
- `nr_site` integer representing the site

CHAPTER 4

Trouble Shooting

I found a bug or have a feature request. How can I get in touch ? Please post issues here or via email mjhoffmann .at. gmail .dot. com or via twitter @maxjhoffmann

My rate constant expression doesn't work. How can I debug it? When initializing the model, the backend uses *kmos.evaluate_rate_expression*. So you can try

```
from kmos import evaluate_rate_expression
evaluate_rate_expression('<your-string-here'>, parameters={})
```

where parameters is a dictionary defining the variable that are defined in the context of the expression evaluation, like so

```
parameters = {'T': {'value': 500},
              'p_NClgas': {'value': 1},
              }
```

Test only parts of your expression to localize the error. Typical mistakes are syntax errors (e.g. unclosed parentheses) and forgotten conversion factors (e.g. eV) which can easily lead to overflow if written in the exponent.

How can I print the chemical potential value, that kmos is using internally? You can then print the explicit value for specific conditions in *kmos shell*, for example like so

```
from kmos import evaluate_rate_expression
print(
    evaluate_rate_expression('mu_COgas',
        {'T':{'value':600},
         'p_COgas': {'value':1}
         }
         )))
```

where 'CO' should be replaced by whatever gas species you are inspecting. And the resulting number is given in eV. kmos linearly interpolates the gas phase chemical potential from the NIST JANAF thermochemical tables if you have downloaded them manually. If you don't have them installed, an error message should get raised which explains how to do so.

When I use *kmos shell* the model doesn't have the species and sites I have defined. Note that Fortran is case-insensitive. Therefore f2py turns all variable and functions names into lower case by convention. Try to lower-case your species or site name.

When I run kmos the GUI way and close it, it seems to hang and I need to use the window manager to kill it. This is a bug waiting to be fixed. To avoid it close the window showing the atoms object by clicking on its close button or Alt-F4 or whichever shortcut your WM uses.

Running a model it sometimes prints *Warning: numerical precision too low, to resolve time-steps* This means that the kMC step of the current process was so small compared to the current kMC time that for the processor $t + \Delta t = t$. This should under normal circumstances only occur if you changed external conditions during a kMC run.

Otherwise it could mean that your rate constants vary over 12 or more orders of magnitude. If this is the case one needs to wonder whether non-coarse graind kMC is actually the right approach for the system. On the hand because the selection of the next process will no longer be reliable and second because reasonable sampling of all involved process may no longer happen.

When running a model without GUI evaluation steps seem very slow. If you have a *kmos.run.KMC_Model* instance and call *model.get_atoms()* the generation of the real-space geometry takes the longest time. If you only have to evaluate coverages or turn-over frequencies you are better off using *model.get_atoms(geometry=False)*, which returns an object with all numbers but without the actual geometry.

What units is kmos using ? By default length are measured in angstrom, energies in eV, pressure in bar, constants are taken from CODATA 2010. Note that the rate expressions though contain explicit conversion factors like *bar*, *eV* etc. If in doubt check the resulting rate constants by hand.

How can I change the occupation of a model at runtime? This is explained in detail at manipulate_model_runtime though the import bit is that you call

```
model._adjust_database()
```

after changing the occupation and before doing the next kMC step.

How can I quickly obtain k_tot ?

That is :: model.base.get_accum_rate(model.proclist.nr_of_proc)

How can I check the system size ?

You can check :: model.lattice.system_size

to get the number of unit cell in the x, y, and z direction. The number of sites per unit cell is stored in

```
model.lattice.spuck
```

a.k.a Sites Per Unit Cell Konstant :-). Or you check

```
model.base.get_volume()
```

to get the total number of sites, i.e. :: model.base.get_volume() == model.lattice.system_size.prod()*model.lattice.spuck => True

More to follow.

Todo

Explain *post-mortem* procedure

CHAPTER 5

Frequently Asked Questions

What other kMC codes are there? Kinetic Monte Carlo codes that I am currently aware of, that are in some form released on the intertubes are with no claim of completeness :

- akmc (G. Henkelman)
- Carlos (J. Lukkien)
- chimp (D. Dooling)
- KMCLib (M. Leetma)
- mapkmc (D. Vlachos)
- Monty (SXM Boerrrigter)
- MoCKa (L. Kunz)
- NASCAM (S. Lucas)
- Spparks (S. Plimpton)
- Zacros (M. Stamatakis)

Though The Google might find you some more. Please drop me a line if you find any information inaccurate.

What does kmos stand for anyways? Good question, initially kmos was supposed to stand for *kinetic modeling on steroids* because it feels really fast to model with kmos. But that confused people too much since we are not modelling reactions on steroids but on surfaces :-). Some other popular variants are

- kMC Modeling Of Surfaces
- kMC modeling offering source
- kMC models on screen

I am open for suggestions.

This document was generated Jan 28, 2016.

k

Symbols

A

B

C

D

E

G

I

J

K

L

M

O

P

S

T

V

W