
kmcos Documentation

Release 1.1.0

Max J. Hoffmann

Apr 12, 2023

Contents

1	Installation	3
1.1	Making a Python Virtual Environment for kmcos within Ubuntu	3
1.2	Installing kmcos on Ubuntu Linux	4
1.3	Installing kmcos on Fedora Linux (typically inside a virtual environment)	5
1.4	Installation on openSUSE 12.1 Linux (Deprecated Instructions)	5
1.5	Installation on openSUSE 13.1 Linux (Deprecated Instructions)	6
1.6	Installation on Mac OS X 10.10 or above (Deprecated Instructions)	6
1.7	Installation on windows	7
1.8	Installing JANAF Thermochemical Tables	7
2	Tutorials	9
2.1	Introduction	9
2.2	The Runtime View	10
2.3	A first kMC Model—the API way	10
2.4	Running the Model From Runfiles	16
2.5	Development	20
2.6	The Model Editor (Deprecated – glade migration is required to revive this feature)	22
3	Topic Guides	27
3.1	The Concept of Kinetic Monte Carlo	27
3.2	Modelling Workflows	29
3.3	The kmcos data model	30
3.4	How the kmcos kMC algorithm works	31
3.5	Temporal acceleration	34
3.6	The otf Backend	35
3.7	The Process Syntax	38
3.8	The Site/Coordinate Syntax	40
3.9	Developer’s guide	42
4	Reference	67
4.1	Model running commands	67
4.2	Connected Variables	76
4.3	Data Types	76
4.4	Editor frontend	81
4.5	Runtime frontend	81
4.6	kmcos kMC project DTD	83
4.7	Backends	85

4.8	Command Line Interface (CLI)	120
5	Trouble Shooting	123
6	Frequently Asked Questions	125
	Python Module Index	127
	Index	129

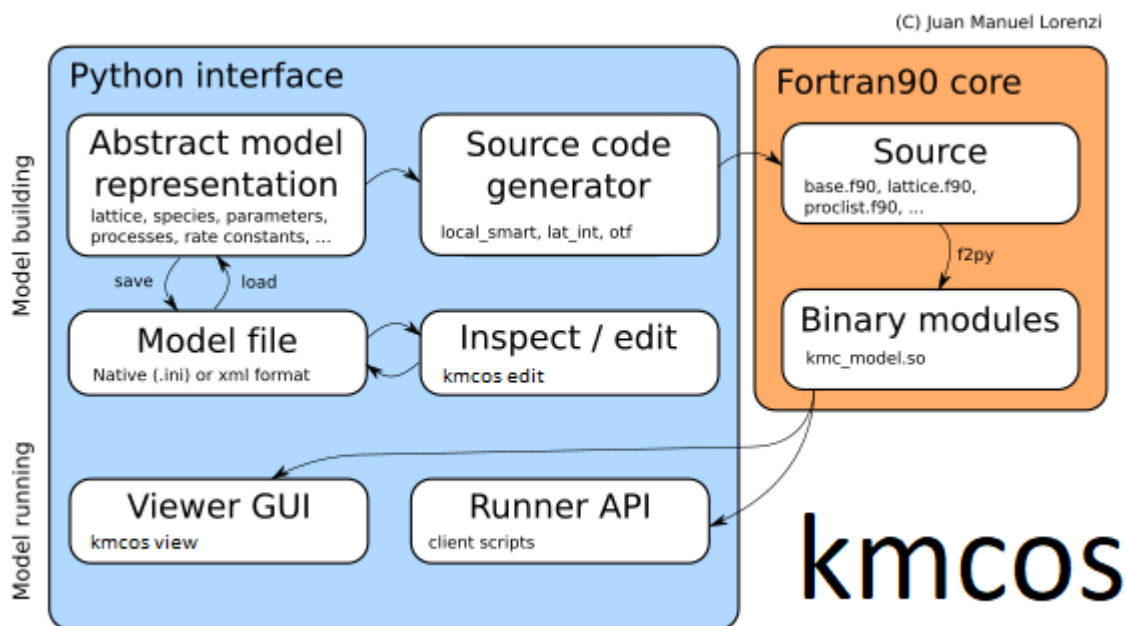


Fig. 1: Things you can do with kmcos.

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

kmcos 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 kmcos. kmcos can be an Editor, an API, a viewer. However all in all kmcos wants to save time filled with repetitive labor and enlarge your stride.

Not sure how to begin? Start with the [API tutorial](#).

CHAPTER 1

Installation

KmcOS has some non-python dependencies so cannot be installed with only pip. It is recommended to install kmcOS on Ubuntu within a python virtual environment, and our instructions are written accordingly. If you plan to use a windows machine, it is recommended to first get [VirtualBox](#) and to [make an Ubuntu virtualmachine](#).

1.1 Making a Python Virtual Environment for kmcOS within Ubuntu

Using a virtual python environment for both installation and for simulations avoids python software conflicts. Here are instructions for installing a python virtual environment.

OPTION 1 (python3-venv):

```
cd ~
sudo apt-get update
sudo apt-get install python3
sudo apt-get install python3-venv
python3 -m pip install --upgrade pip
python3 -m venv ~/VENV/kmcOS
source ~/VENV/kmcOS/bin/activate
```

To use kmcOS after this installation, you will need to use that source activation command from the terminal each time. When finished, you can exit this virtualenv by typing 'deactivate'.

OPTION 2 (virtualenv):

```
cd ~
sudo apt-get update
sudo apt-get install python3
sudo apt-get install virtualenv
python3 -m pip install --upgrade pip
virtualenv -p /usr/bin/python3 ~/VENV/kmcOS #If this fails, try typing "which python3
↪ " and replace the path "/usr/bin/python3" with what your system provides.
source ~/VENV/kmcOS/bin/activate
```

To use kmcos after this installation, you will need to use that source activation command from the terminal each time. When finished, you can exit this virtualenv by typing ‘deactivate’. Though you should not need it, you can find more information on virtualenv at [this video](#) and [the official website](#)

OPTION 3 (anaconda): If you will be installing kmcos in an anaconda environment, you can make a new environment named ‘kmcos’ from anaconda navigator. See for example [this link](#) .

Virtual environment installations do not require the “-user” tag as the python packages are ‘sandboxed’ during installation. Accordingly, the “-user” tags are commented out in our further instructions.

1.2 Installing kmcos on Ubuntu Linux

If you are a typical user, first make sure you are in your virtual environment (after preparation by the above instructions):

```
source ~/VENV/kmcos/bin/activate
```

The easiest way to install kmcos is to use one of the automatic installers:

```
cd ~
sudo apt-get install git
git clone https://github.com/kmcos/kmcos-installers
cd kmcos-installers
python3 -m pip install --upgrade pip
bash install-kmcos-linux-venv.bash #use 'bash install-kmcos-linux-user.bash' if you
↪are not using a venv. #For the develop branch, use install-kmcos-linux-venv-
↪develop.bash or install-kmcos-linux-user-develop.bash
```

For personal computer usage (not on a supercomputer), it is a good idea to also run the following command, which will add the kmcos viewer and movie maker:

```
bash install-kmcos-complete-linux-venv-Ubuntu20.bash #this is for Ubuntu20. There is
↪also an Ubuntu18 version.
```

If everything has gone well, you have a minimal installation completed! And now you are done and can leave this installation page!

If the above simple way does not work for you, you will need to go through the commands manually one at a time from [installation on a venv](#) or [installation as a user](#) . A kmcosInstallation directory is created during installation. The files in the kmcosInstallation are no longer needed after installation, but it has examples in it. So you can navigate into that directory and go through the examples, or you can remove the kmcosInstallation directory using ‘rm -r directoryname’.

When doing kmcos upgrades, you will not need to use git again. For kmcos upgrades, you can just use the earlier pip command:

```
pip3 install kmcos[MINIMAL] --upgrade #--user
```

(Optional) If you would like to use the kmcos view capability, you will need to install some non-python dependencies and then kmcos complete:

```
sudo apt-get install python-ase
sudo apt-get install python3-gi
pip3 install ase #--user
pip3 install kmcos[COMPLETE] --upgrade #--user
```


If the last command of ‘`pip3 install kmcOS[COMPLETE] --upgrade --user`’ gives an error before finishing, try the command a second time.

1.3 Installing kmcOS on Fedora Linux (typically inside a virtual environment)

Install development tools gcc and fortran.

For fedora 32+

```
sudo dnf groupinstall "Development Tools" "Development Libraries"
sudo dnf install gcc-gfortran
```

For fedora below 32

```
sudo dnf groupinstall @development-tools @development-libraries
sudo dnf install gcc-gfortran
```

Make a virtual environment for the kmcOS and activate it:

```
python3 -m venv ~/VENV/kmcOS
source ~/VENV/kmcOS/bin/activate
```

Clone the kmcOS github repository in a folder you want and change to the kmcOS directory:

```
git clone https://github.com/kmcOS/kmcOS.git
cd kmcOS
```

Install the python package requirements and finally the kmcOS package:

```
pip3 install numpy lxml ase matplotlib UnitTesterSG CiteSoft IPython
python3 setup.py install
```

1.4 Installation on openSUSE 12.1 Linux (Deprecated Instructions)

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 [GitLab website](#) unzip it and install it with:

```
python setup.py install
```

1.5 Installation on openSUSE 13.1 Linux (Deprecated Instructions)

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](#) .

1.6 Installation on Mac OS X 10.10 or above (Deprecated Instructions)

There is more than one way to get required dependencies. MacPorts was previously tested and worked.

As of 2022, the MacPorts way does not seem to be working and the virtual machine way is recommended.

The Virtual Machine Way:

Needed to use Ubuntu 20.04 (Using Ubuntu 22 did not work).

Guest additions was not working on the mac. So needed to do below in addition to the instructions in the intro2kmcOS doc.

- 1) Needed to find Virtual Box with finder, right click on the Virtual Box application, show files / show contents, needed to find the VirtualBox.iso file, copy it out to a regular MacOS directory.
- 2) Perl was not working, so needed to do the following:

```
sudo apt-get update
sudo apt-get install build-essential gcc make perl dkms
```

That worked, then rebooted Ubuntu.

- 3) Navigated to the virtual disc of the guest additions CD (virtual compact disc):

```
bash autorun.sh
```

Then was able to use the virtual machine as well as install kmcOS normally.

The MacPorts Way:

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. kmcOS 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 port install -v py27-ipython
sudo port select --set ipython py27-ipython

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 ...
```

(continues on next page)

(continued from previous page)

```
# 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-kmcOS --user
```

1.7 Installation on windows

Direct installation on windows is currently not supported, but it is possible to use either “WSL” or to use Ubuntu on a virtualbox. It is recommended to download virtualbox, to install Ubuntu, and then follow the Ubuntu installation instructions in the intro2kmcOS pdf file here: <https://github.com/kmcOS/intro2kmcOS>. You may need to adjust the resolution to work effectively.

If you prefer to use WSL rather than Virtualbox, you will need to install WSL Ubuntu. Press the “start menu” button. Type “Windows Powershell” but don’t press enter: Use run as administrator. Then enter:

```
wsl --install -d Ubuntu
```

Now, you can close the Powershell window. Within ubuntu, use:

```
sudo apt update
sudo apt install x11-apps
```

From the terminal, type:

```
xeyes &
```

With windows 11 and higher, you may see a GUI pop up. If you do not, then you probably will not be able to use a GUI with WSL, and the kmcOS export_movie feature also will not work.

For future reference: “cd ~” will take you to the home (default) place for working in WSL Ubuntu, while “cd /” will take you to the root directory of WSL Ubuntu.

For sharing files, “cd /mnt/c” will let you access files on to go to the windows C drive. By going to mnt/c, you can move files back and forth between Ubuntu directories and the Windows directories.

Now that you have WSL working with Ubuntu, follow the regular instructions from the top of this Installation page. Going forward, you can start WSL Ubuntu by finding Ubuntu in the windows start menu.

1.8 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.

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 *jana_f_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

2.1 Introduction

kmcos is designed for lattice based Kinetic Monte Carlo simulations to understand chemical kinetics and mechanisms. It has been used to produce more than 10 scientific publications. The best way to learn how to use kmcos is by following the examples.

If you have already followed the kmcos installation instructions and still have the kmcosInstallation directory, then navigate to /kmcosInstallation/kmcos/examples

If you do not have that directory, but have kmcos installed, go to <https://github.com/kmcos/kmcos> Click on the green button and download zip, to get the examples.

Inside /examples/, run the following commands

```
python3 MyFirstSnapshots__build.py
cd MyFirstSnapshots_local_smart
python3 runfile.py
```

The first command uses a python file to create a chemical model (process definitions) and a KMC modeling executable as well. The “local_smart” is the default backend (default “KMC Engine”, kmcos has several).

After the simulation has run, you will see a csv file named runfile_TOFs_and_Coverages.csv, open this file to see your first KMC output!

Various examples exist. More features and a thorough tutorial are forthcoming. Please join the kmcos-users group <https://groups.google.com/g/kmcos-users> and email any questions if you get stuck.

2.1.1 Feature overview

This paragraph is from __init__.py

With kmcos you can:

- easily create and modify kMC models through GUI

- store and exchange kMC models through XML
- generate fast, platform independent, self-contained code¹
- run kMC models through GUI or python bindings

kmcos 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.

kmcos' 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.

Typical users will run kmcos entirely from python code by following the examples.

2.2 The Runtime View

2.3 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.

2.3.1 Build the model

You may also look at `MyFirstDiffusion__build.py` in the examples directory.

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

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

which imports all classes that make up a kMC project. The functions from *kmcos.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). First you should instantiate a new project and fill in meta information

```
kmc_model = kmcos.create_kmc_model()
kmc_model.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

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

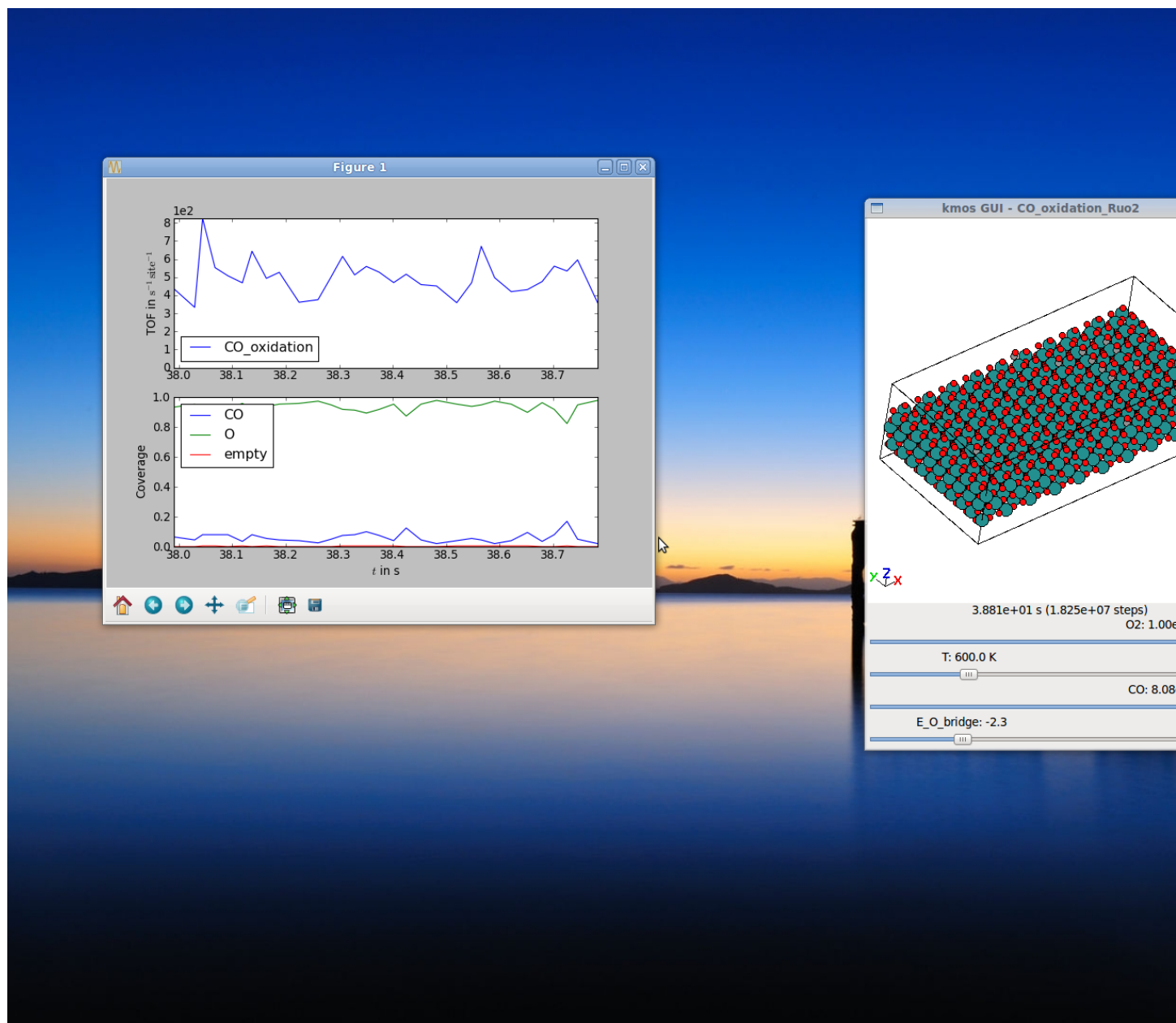


Fig. 1: The compiled module can be run and watched in realtime. When parameters are changed this is immediately reflected in the rate constants.

```
kmc_model.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

```
kmc_model.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 = kmc_model.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

```
kmc_model.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*.

2.3.2 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 loses 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:

```
kmc_model.add_parameter(name='T', value=600., adjustable=True, min=400, max=800)  
kmc_model.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 behave later in the runtime GUI.

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

```
kmc_model.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

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

To define processes we first need a coordinate³

```
coord = kmc_model.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

```
kmc_model.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 kmcos 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 $1/(k_boltzmann*T)$ 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:

```
kmc_model.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, kmcos also knows a shorthand notation for processes. In order to produce the same process you could also type

```
kmc_model.parse_process('CO_desorption; CO@hollow->empty@hollow ; p_CO*bar*A/
↪sqrt(2*pi*umass*m_CO/beta)*exp(beta*deltaG*eV)')
```

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

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

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

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.

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

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

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

```
"<site-name>.<offset>.<lattice>"
```

check *Manual generation* for details.

2.3.3 Export, save, compile

Before we compile the model, we should specify and understand the various backends that are involved.

local_smart backend (default) for models with <100 processes. lat_int backend for models with >100 processes. (build the model same ways local_smart but different backend for compile step) of backend requires custom model (build requires different process definitions compared to local_smart) and can work for models which require >10000 processes, since each process rate is calculated on the fly instead of being held in memory.

Here is how we specify the model's backend

```
kmc_model.backend = 'local_smart'  
kmc_model.backend = 'lat_int'  
kmc_model.backend = 'otf'
```

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

```
kmc_model.save_model()  
kmcos.compile(kmc_model)
```

This creates an XML file with the full definition of your model and exports the model to compiled code.

Now is the time to leave the python shell. In the current directory you should see a *myfirst_kmc.xml*. You will also see a directory ending with *_local_smart*, this directory includes your compiled model.

You can also skip the model exporting (and do it later) by commenting out `kmcos.compile(kmc_model)`: then you can use a separate python file later. For some installations, you can use *kmcOS export myfirst_kmc.xml* from the linux terminal when you are in the same directory as the XML.

During troubleshooting, exporting separately can sometimes be useful to make sure the compiling occurs gracefully without any line containing an error.

Running and viewing the model

If you now *cd* to that folder *myfirst_kmc_local_smart* and run

```
python3 kmc_settings.py benchmark
```

You should see that the model was able to run! Next, let's try seeing how it looks visually with

```
python3 kmc_settings.py view
```

The "view" command only works on certain operating systems. For some installations, one can alternatively type *kmcOS benchmark* and *kmcOS view*.

For running the model, it is recommended to use a runfile.

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 kmcOS approach and methods you should into [topic guides](#).

In technical terms, kmcOS is run an API via the kmcOS python module.

Additionally, though now discouraged, kmcOS can be invoked directly from the command line in one of the following ways:

```
kmcOS [help] (all|benchmark|build|edit|export|help|import|rebuild|run|settings-
↪export|shell|version|view|xml) [options]
```

2.3.4 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.

2.3.5 An alternative way using .ini files

Presently, a full description of the .ini capability is not being provided because this way is not the standard way of using kmcOS. However, it is available. This method is an alternative to making an xml file, and can be used instead of kmcOS export.

Prepare a minimal input file with the following content and save it as `mini_101.ini`

```
[Meta]
author = Your Name
email = you@server.com
model_dimension = 2
model_name = fcc_100

[Species empty]
color = #FFFFFF

[Species CO]
representation = Atoms("CO", [[0, 0, 0], [0, 0, 1.17]])
color = #FF0000

[Lattice]
cell_size = 3.5 3.5 10.0

[Layer simple_cubic]
site_hollow = (0.5, 0.5, 0.5)
color = #FFFFFF

[Parameter k_CO_ads]
value = 100
adjustable = True
min = 1
max = 1e13
scale = log
```

(continues on next page)

(continued from previous page)

```
[Parameter k_CO_des]
value = 100
adjustable = True
min = 1
max = 1e13
scale = log

[Process CO_ads]
rate_constant = k_CO_ads
conditions = empty@hollow
actions = CO@hollow
tof_count = {'adsorption':1}

[Process CO_des]
rate_constant = k_CO_des
conditions = CO@hollow
actions = empty@hollow
tof_count = {'desorption':1}
```

In the same directory run `kmcOS export mini_101.ini`. You should now have a folder `mini_101_local_smart` in the same directory. `cd` into it and run `kmcOS benchmark`. If everything went well you should see something like

```
Using the [local_smart] backend.
1000000 steps took 1.51 seconds
Or 6.62e+05 steps/s
```

In the same directory try running `kmcOS view` to watch the model run or fire up `kmcOS shell` to interact with the model interactively. Explore more commands with `kmcOS help` and please refer to the documentation how to build complex model and evaluate them systematically. To test all bells and whistles try `kmcOS edit mini_101.ini` and inspect the model visually.

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

2.4 Running the Model From Runfiles

2.4.1 Running the Model—the API way

Normally, one uses python runfiles. However, it is convenient to initially run commands interactively for learning purposes. The simplest thing to do is to start the model from within a compiled model directory using “python3 `kmc_settings.py run`”

That will start a python shell, allowing one to skip the below commands

```
#!/usr/bin/env python
from kmcOS.run import KMC_Model
model = KMC_Model()
```

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

```
%logstart some_scriptname.py
```

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

When using a runfile, the starting banner can be turned off by using:

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

Now that you have got a model, you try to do some KMC steps

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

which would run 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
```

If one wants to know what the next kmc step will be and at which site, without executing the step, one can use

```
model.get_next_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 attributes 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 is finished as this frees the memory allocated by the Fortran modules. This is particularly necessary if you want to run more than one simulation in one script.

2.4.2 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 kmcOS.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.

2.4.3 Manipulating the Model Species at Runtime

To change species on the lattice at the start of simulation or at any other time in the simulation, one can change either the whole configuration, or only species on a specific site.

To change species on a specific site, one uses the put command. There are several syntaxes to use the put command

```
model.put(site=[x,y,z,n], model.proclist.<species>)
Where 'n' and <species> are the site type and species, respectively. For example:
model.put([0,0,0,model.lattice.ruo2_bridge], model.proclist.co)
model.put([0,0,0,"ruo2_bridge"], "model.proclist.co")
model.put([0,0,0,2], 1) #The 'n' is has indexing starting from 1 (there is no 0 for_
↪n), whereas the <species> indexing starts at 0.
```

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

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

note that after using ‘_put’, one must remember to call *_adjust_database()* before executing any next step or the database of available processes will not match the species, the kmc simulation will become incorrect and likely crash after some steps.

2.4.4 Saving and Reloading the State of the Simulation

If one wants to set the whole configuration of the lattice once can retrieve it, save it, and load it with the following commands

```
model.dump_config("YourConfigurationName")
model.load_config("YourConfigurationName")
```

While it is not necessary for a regular user to know, those commands use the following internal commands as part of how they function

```
#saving the configuration uses:
config = model._get_configuration()
#loading configuration uses:
model._set_configuration(config)
model._adjust_database()
```

However, simply saving and loading the configuration will not allow you to exactly reproduce the simulation where it left off. To do that, you also need to save and reload the pseudo random generator’s state

```
PRNG_state = model.proclist.get_seed().tolist() #This list can be saved as a pickle_
↪or in a text file.
model.proclist.put_seed(PRNG_state) #This command takes the PRNG_state as a list and_
↪inputs into the simulation.
```

By saving both the configuration and the PRNG_state, one can start a simulation again on the same trajectory (providing one sets the parameters such as temperature and pressure). The snapshots module includes methods saving and loading the configuration, PRNG_state, and parameters. A single command to save all aspects of the simulation and reload the simulation where it leftoff will later be added into the main code and added to the tutorials.

2.4.5 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. kmcOS certainly cannot parallelize a single system over processors. However one can run several kmcOS 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 separately at once, for example to scan some set of parameters on a multicore computer. This kind of problem can be considered *embarrassingly 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 *kmcOS* we need to import *multiprocessing*

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

and let's say you wanted to scan a range of temperature, while keeping all other parameters 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 kmcOS.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()
```

2.5 Development

Contributions of any sort are of course quite welcome. It is best to first contact the developers. After that, patches and comments are ideally sent in form of email, pull request, or github issues.

Below is advice from the original developer:

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
```

To make testing and comparison even easier it would be helpful if you create an account with [Travis CI](#) and run your commits through the test suite.

Have a look at Google's Python [style guide](#) as far as style questions go.

2.5.1 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

```
python3 kmcos view
```

command from the command line. For this two work you need to be in the same directory as these two files (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.

2.5.2 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 an 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 kmcos.run import KMC_Model
from kmcos.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

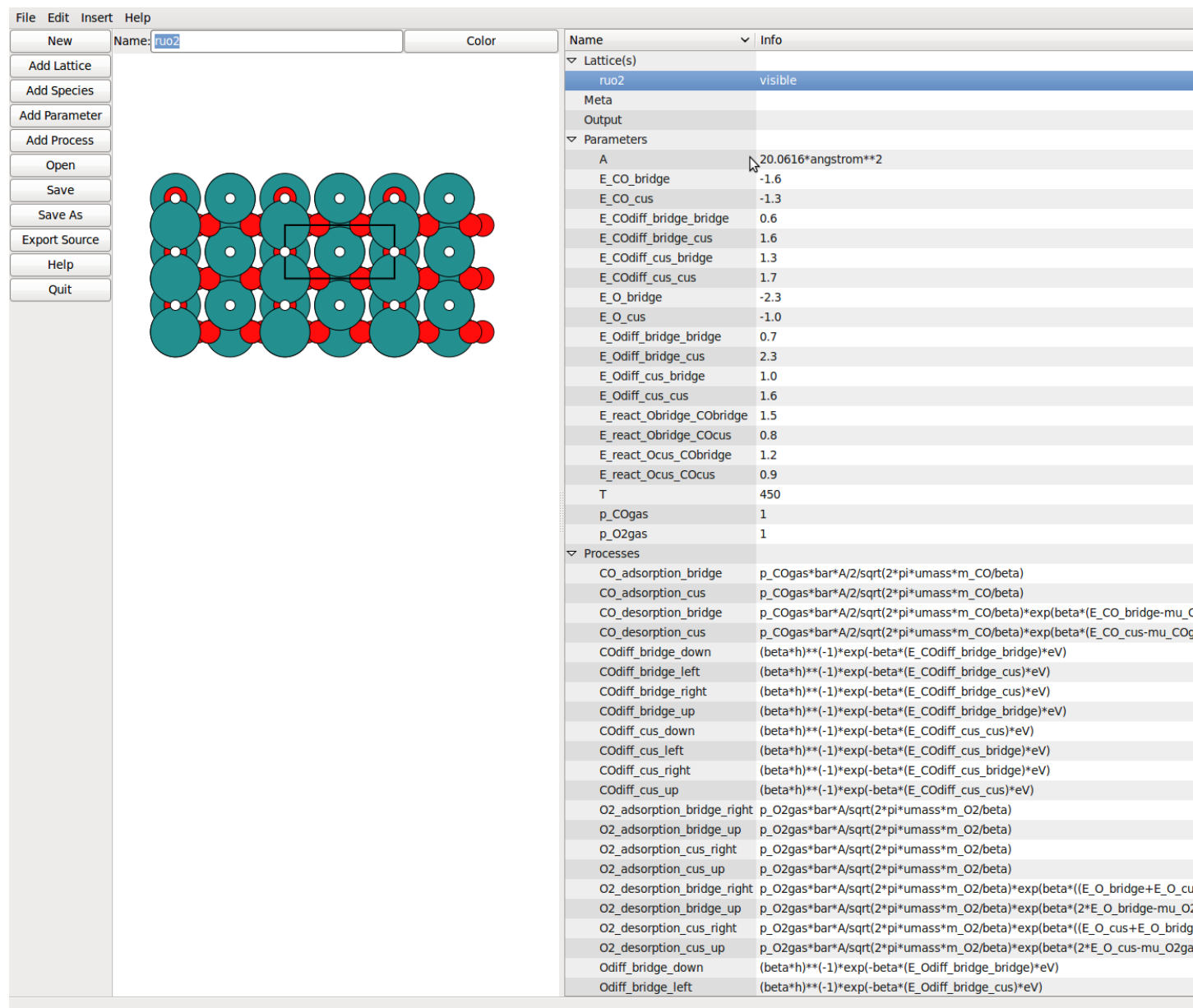
```
kmcos view, run, or benchmark
```

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.

2.6 The Model Editor (Deprecated – glade migration is required to revive this feature)



The screenshot displays the Model Editor interface. On the left, a vertical toolbar contains buttons for 'New', 'Add Lattice', 'Add Species', 'Add Parameter', 'Add Process', 'Open', 'Save', 'Save As', 'Export Source', 'Help', and 'Quit'. The main window is divided into two panes. The left pane shows a 2D lattice of sites, represented by teal and red circles, with a black box highlighting a specific site. The right pane is a table with columns 'Name' and 'Info'.

Name	Info
ruo2	visible
Meta	
Output	
Parameters	
A	20.0616*angstrom**2
E_CO_bridge	-1.6
E_CO_cus	-1.3
E_CODiff_bridge_bridge	0.6
E_CODiff_bridge_cus	1.6
E_CODiff_cus_bridge	1.3
E_CODiff_cus_cus	1.7
E_O_bridge	-2.3
E_O_cus	-1.0
E_Odiff_bridge_bridge	0.7
E_Odiff_bridge_cus	2.3
E_Odiff_cus_bridge	1.0
E_Odiff_cus_cus	1.6
E_react_Obridge_CObridge	1.5
E_react_Obridge_COcus	0.8
E_react_Ocus_CObridge	1.2
E_react_Ocus_COcus	0.9
T	450
p_COgas	1
p_O2gas	1
Processes	
CO_adsorption_bridge	$p_{\text{COgas}} \cdot \bar{A} / \sqrt{2 \cdot \pi \cdot u_{\text{mass}} \cdot m_{\text{CO}} / \beta}$
CO_adsorption_cus	$p_{\text{COgas}} \cdot \bar{A} / \sqrt{2 \cdot \pi \cdot u_{\text{mass}} \cdot m_{\text{CO}} / \beta}$
CO_desorption_bridge	$p_{\text{COgas}} \cdot \bar{A} / \sqrt{2 \cdot \pi \cdot u_{\text{mass}} \cdot m_{\text{CO}} / \beta} \cdot \exp(\beta \cdot (E_{\text{CO_bridge}} - \mu_{\text{CO}}))$
CO_desorption_cus	$p_{\text{COgas}} \cdot \bar{A} / \sqrt{2 \cdot \pi \cdot u_{\text{mass}} \cdot m_{\text{CO}} / \beta} \cdot \exp(\beta \cdot (E_{\text{CO_cus}} - \mu_{\text{CO}}))$
CODiff_bridge_down	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot (E_{\text{CODiff_bridge_bridge}}) \cdot eV)$
CODiff_bridge_left	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot (E_{\text{CODiff_bridge_cus}}) \cdot eV)$
CODiff_bridge_right	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot (E_{\text{CODiff_bridge_cus}}) \cdot eV)$
CODiff_bridge_up	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot (E_{\text{CODiff_bridge_bridge}}) \cdot eV)$
CODiff_cus_down	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot (E_{\text{CODiff_cus_cus}}) \cdot eV)$
CODiff_cus_left	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot (E_{\text{CODiff_cus_bridge}}) \cdot eV)$
CODiff_cus_right	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot (E_{\text{CODiff_cus_bridge}}) \cdot eV)$
CODiff_cus_up	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot (E_{\text{CODiff_cus_cus}}) \cdot eV)$
O2_adsorption_bridge_right	$p_{\text{O2gas}} \cdot \bar{A} / \sqrt{2 \cdot \pi \cdot u_{\text{mass}} \cdot m_{\text{O2}} / \beta}$
O2_adsorption_bridge_up	$p_{\text{O2gas}} \cdot \bar{A} / \sqrt{2 \cdot \pi \cdot u_{\text{mass}} \cdot m_{\text{O2}} / \beta}$
O2_adsorption_cus_right	$p_{\text{O2gas}} \cdot \bar{A} / \sqrt{2 \cdot \pi \cdot u_{\text{mass}} \cdot m_{\text{O2}} / \beta}$
O2_adsorption_cus_up	$p_{\text{O2gas}} \cdot \bar{A} / \sqrt{2 \cdot \pi \cdot u_{\text{mass}} \cdot m_{\text{O2}} / \beta}$
O2_desorption_bridge_right	$p_{\text{O2gas}} \cdot \bar{A} / \sqrt{2 \cdot \pi \cdot u_{\text{mass}} \cdot m_{\text{O2}} / \beta} \cdot \exp(\beta \cdot ((E_{\text{O_bridge}} + E_{\text{O_cus}}) - \mu_{\text{O2}}))$
O2_desorption_bridge_up	$p_{\text{O2gas}} \cdot \bar{A} / \sqrt{2 \cdot \pi \cdot u_{\text{mass}} \cdot m_{\text{O2}} / \beta} \cdot \exp(\beta \cdot (2 \cdot E_{\text{O_bridge}} - \mu_{\text{O2}}))$
O2_desorption_cus_right	$p_{\text{O2gas}} \cdot \bar{A} / \sqrt{2 \cdot \pi \cdot u_{\text{mass}} \cdot m_{\text{O2}} / \beta} \cdot \exp(\beta \cdot ((E_{\text{O_cus}} + E_{\text{O_bridge}}) - \mu_{\text{O2}}))$
O2_desorption_cus_up	$p_{\text{O2gas}} \cdot \bar{A} / \sqrt{2 \cdot \pi \cdot u_{\text{mass}} \cdot m_{\text{O2}} / \beta} \cdot \exp(\beta \cdot (2 \cdot E_{\text{O_cus}} - \mu_{\text{O2}}))$
Odiff_bridge_down	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot (E_{\text{Odiff_bridge_bridge}}) \cdot eV)$
Odiff_bridge_left	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot (E_{\text{Odiff_bridge_cus}}) \cdot eV)$

Fig. 2: The lattice view allows to define sites by simple pointing.

The screenshot displays the kmcos Model Editor interface. On the left, a sidebar contains buttons for 'New', 'Add Lattice', 'Add Species', 'Add Parameter', 'Add Process', 'Open', 'Save', 'Save As', 'Export Source', 'Help', and 'Quit'. The main area is divided into two panels. The top panel shows the configuration for a new parameter: 'Name' is 'i', 'Value' is '450', and 'adjustable' is checked. Below this, 'min' is '300.0' and 'max' is '1,500.0'. The bottom panel is a table listing model parameters and their values.

Name	Info
▼ Lattice(s)	
ruo2	visible
Meta	
Output	
▼ Parameters	
A	20.0616*angstrom**2
E_CO_bridge	-1.6
E_CO_cus	-1.3
E_CODiff_bridge_bridge	0.6
E_CODiff_bridge_cus	1.6
E_CODiff_cus_bridge	1.3
E_CODiff_cus_cus	1.7
E_O_bridge	-2.3
E_O_cus	-1.0
E_Odiff_bridge_bridge	0.7
E_Odiff_bridge_cus	2.3
E_Odiff_cus_bridge	1.0
E_Odiff_cus_cus	1.6
E_react_Obridge_CObridge	1.5
E_react_Obridge_COcus	0.8
E_react_OCus_CObridge	1.2
E_react_OCus_COcus	0.9
T	450
p_COgas	1
p_O2gas	1
▼ Processes	
CO_adsorption_bridge	$p_{COgas} \cdot \bar{A} / \sqrt{2 \cdot \pi \cdot u_{mass} \cdot m_{CO} / \beta}$
CO_adsorption_cus	$p_{COgas} \cdot \bar{A} / \sqrt{2 \cdot \pi \cdot u_{mass} \cdot m_{CO} / \beta}$
CO_desorption_bridge	$p_{COgas} \cdot \bar{A} / \sqrt{2 \cdot \pi \cdot u_{mass} \cdot m_{CO} / \beta} \cdot \exp(\beta \cdot (E_{CO_bridge} - \mu_{CO}))$
CO_desorption_cus	$p_{COgas} \cdot \bar{A} / \sqrt{2 \cdot \pi \cdot u_{mass} \cdot m_{CO} / \beta} \cdot \exp(\beta \cdot (E_{CO_cus} - \mu_{CO}))$
CODiff_bridge_down	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot (E_{CODiff_bridge_bridge})) \cdot eV$
CODiff_bridge_left	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot (E_{CODiff_bridge_cus})) \cdot eV$
CODiff_bridge_right	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot (E_{CODiff_bridge_cus})) \cdot eV$
CODiff_bridge_up	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot (E_{CODiff_bridge_bridge})) \cdot eV$
CODiff_cus_down	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot (E_{CODiff_cus_cus})) \cdot eV$
CODiff_cus_left	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot (E_{CODiff_cus_bridge})) \cdot eV$
CODiff_cus_right	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot (E_{CODiff_cus_bridge})) \cdot eV$
CODiff_cus_up	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot (E_{CODiff_cus_cus})) \cdot eV$
O2_adsorption_bridge_right	$p_{O2gas} \cdot \bar{A} / \sqrt{2 \cdot \pi \cdot u_{mass} \cdot m_{O2} / \beta}$
O2_adsorption_bridge_up	$p_{O2gas} \cdot \bar{A} / \sqrt{2 \cdot \pi \cdot u_{mass} \cdot m_{O2} / \beta}$
O2_adsorption_cus_right	$p_{O2gas} \cdot \bar{A} / \sqrt{2 \cdot \pi \cdot u_{mass} \cdot m_{O2} / \beta}$
O2_adsorption_cus_up	$p_{O2gas} \cdot \bar{A} / \sqrt{2 \cdot \pi \cdot u_{mass} \cdot m_{O2} / \beta}$
O2_desorption_bridge_right	$p_{O2gas} \cdot \bar{A} / \sqrt{2 \cdot \pi \cdot u_{mass} \cdot m_{O2} / \beta} \cdot \exp(\beta \cdot ((E_{O_bridge} + E_{O_cus}) - \mu_{O2}))$
O2_desorption_bridge_up	$p_{O2gas} \cdot \bar{A} / \sqrt{2 \cdot \pi \cdot u_{mass} \cdot m_{O2} / \beta} \cdot \exp(\beta \cdot ((E_{O_bridge} + E_{O_cus}) - \mu_{O2}))$
O2_desorption_cus_right	$p_{O2gas} \cdot \bar{A} / \sqrt{2 \cdot \pi \cdot u_{mass} \cdot m_{O2} / \beta} \cdot \exp(\beta \cdot ((E_{O_cus} + E_{O_bridge}) - \mu_{O2}))$
O2_desorption_cus_up	$p_{O2gas} \cdot \bar{A} / \sqrt{2 \cdot \pi \cdot u_{mass} \cdot m_{O2} / \beta} \cdot \exp(\beta \cdot ((E_{O_cus} + E_{O_bridge}) - \mu_{O2}))$
Odifff_bridge_down	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot (E_{Odifff_bridge_bridge})) \cdot eV$
Odifff_bridge_left	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot (E_{Odifff_bridge_cus})) \cdot eV$

Fig. 3: Model parameters can be defined including ranges to vary them over in the runtime viewer.

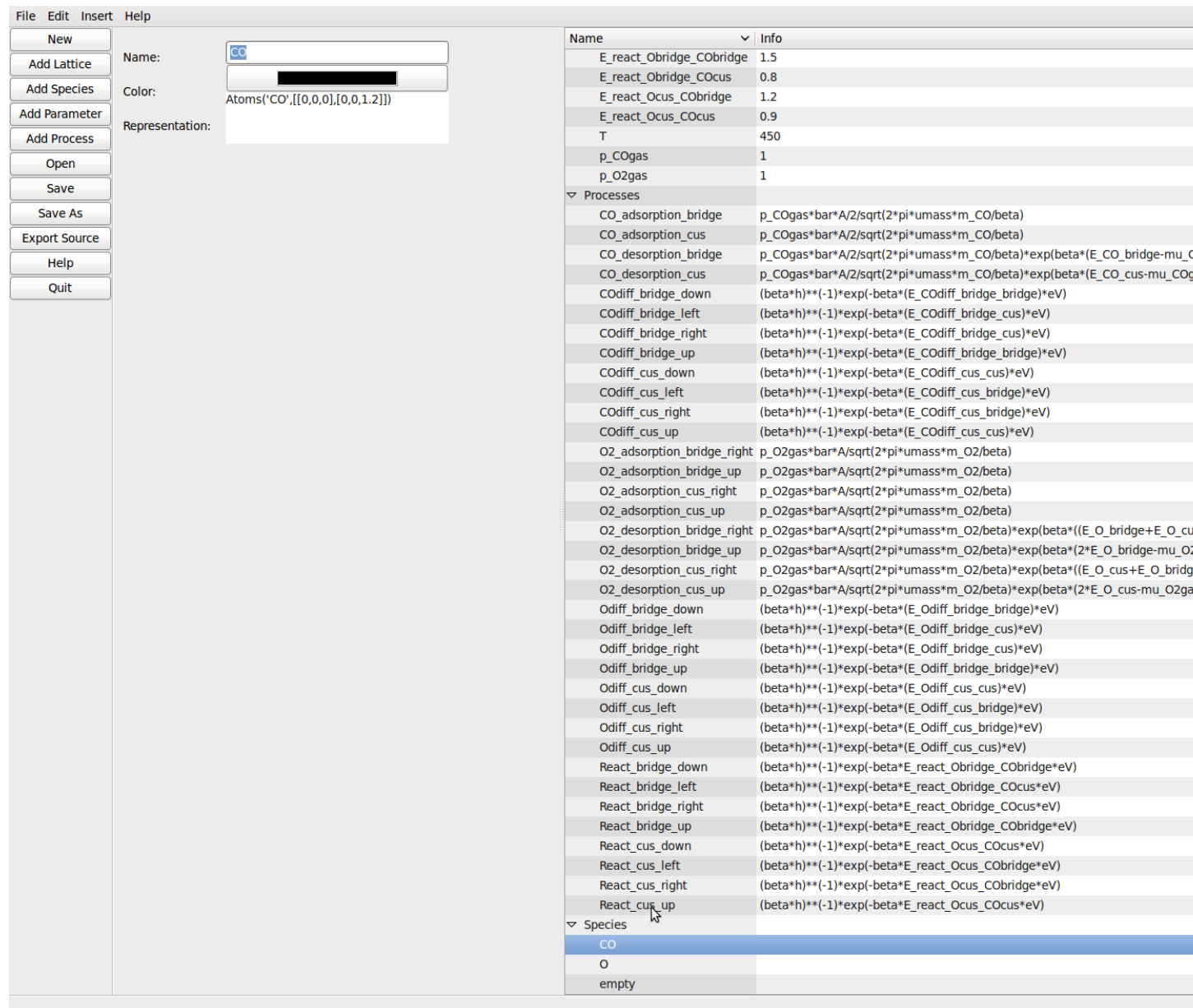


Fig. 4: Species can be added here. The color is used to represent them in the 2D editor view. The string is an ASE atoms constructor for display at runtime.

The screenshot shows the kmcos Model Editor interface. The left panel contains a menu with options: New, Add Lattice, Add Species, Add Parameter, Add Process, Open, Save, Save As, Export Source, Help, and Quit. The main area displays a grid of lattice sites. The top part of the grid is labeled 'Reservoir Area' and contains three colored circles (black, red, white). The bottom part is labeled 'Lattice Area' and contains a grid of sites. Two sites in the center are highlighted with black and white circles. The right panel shows a list of processes and species. The 'Processes' list includes various adsorption, desorption, and diffusion processes with their corresponding rate expressions. The 'Species' list includes CO, O, and empty sites.

Name	Info
E_react_Obridge_CObridge	1.5
E_react_Obridge_COcus	0.8
E_react_Ocus_CObridge	1.2
E_react_Ocus_COcus	0.9
T	450
p_COgas	1
p_O2gas	1
Processes	
CO_adsorption_bridge	$p_{\text{COgas}} \cdot \bar{A} / \sqrt{2 \cdot \pi \cdot m_{\text{CO}} / \beta}$
CO_adsorption_cus	$p_{\text{COgas}} \cdot \bar{A} / \sqrt{2 \cdot \pi \cdot m_{\text{CO}} / \beta}$
CO_desorption_bridge	$p_{\text{COgas}} \cdot \bar{A} / \sqrt{2 \cdot \pi \cdot m_{\text{CO}} / \beta} \cdot \exp(\beta \cdot (E_{\text{CO_bridge}} - \mu_{\text{CO}}))$
CO_desorption_cus	$p_{\text{COgas}} \cdot \bar{A} / \sqrt{2 \cdot \pi \cdot m_{\text{CO}} / \beta} \cdot \exp(\beta \cdot (E_{\text{CO_cus}} - \mu_{\text{CO}}))$
CODiff_bridge_down	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot (E_{\text{CODiff_bridge_bridge}}) \cdot eV)$
CODiff_bridge_left	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot (E_{\text{CODiff_bridge_cus}}) \cdot eV)$
CODiff_bridge_right	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot (E_{\text{CODiff_bridge_cus}}) \cdot eV)$
CODiff_bridge_up	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot (E_{\text{CODiff_bridge_bridge}}) \cdot eV)$
CODiff_cus_down	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot (E_{\text{CODiff_cus_cus}}) \cdot eV)$
CODiff_cus_left	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot (E_{\text{CODiff_cus_bridge}}) \cdot eV)$
CODiff_cus_right	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot (E_{\text{CODiff_cus_bridge}}) \cdot eV)$
CODiff_cus_up	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot (E_{\text{CODiff_cus_cus}}) \cdot eV)$
O2_adsorption_bridge_right	$p_{\text{O2gas}} \cdot \bar{A} / \sqrt{2 \cdot \pi \cdot m_{\text{O2}} / \beta}$
O2_adsorption_bridge_up	$p_{\text{O2gas}} \cdot \bar{A} / \sqrt{2 \cdot \pi \cdot m_{\text{O2}} / \beta}$
O2_adsorption_cus_right	$p_{\text{O2gas}} \cdot \bar{A} / \sqrt{2 \cdot \pi \cdot m_{\text{O2}} / \beta}$
O2_adsorption_cus_up	$p_{\text{O2gas}} \cdot \bar{A} / \sqrt{2 \cdot \pi \cdot m_{\text{O2}} / \beta}$
O2_desorption_bridge_right	$p_{\text{O2gas}} \cdot \bar{A} / \sqrt{2 \cdot \pi \cdot m_{\text{O2}} / \beta} \cdot \exp(\beta \cdot ((E_{\text{O_bridge}} + E_{\text{O_cus}}) - \mu_{\text{O2}}))$
O2_desorption_bridge_up	$p_{\text{O2gas}} \cdot \bar{A} / \sqrt{2 \cdot \pi \cdot m_{\text{O2}} / \beta} \cdot \exp(\beta \cdot (2 \cdot E_{\text{O_bridge}} - \mu_{\text{O2}}))$
O2_desorption_cus_right	$p_{\text{O2gas}} \cdot \bar{A} / \sqrt{2 \cdot \pi \cdot m_{\text{O2}} / \beta} \cdot \exp(\beta \cdot ((E_{\text{O_cus}} + E_{\text{O_bridge}}) - \mu_{\text{O2}}))$
O2_desorption_cus_up	$p_{\text{O2gas}} \cdot \bar{A} / \sqrt{2 \cdot \pi \cdot m_{\text{O2}} / \beta} \cdot \exp(\beta \cdot (2 \cdot E_{\text{O_cus}} - \mu_{\text{O2}}))$
Odifff_bridge_down	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot (E_{\text{Odifff_bridge_bridge}}) \cdot eV)$
Odifff_bridge_left	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot (E_{\text{Odifff_bridge_cus}}) \cdot eV)$
Odifff_bridge_right	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot (E_{\text{Odifff_bridge_cus}}) \cdot eV)$
Odifff_bridge_up	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot (E_{\text{Odifff_bridge_bridge}}) \cdot eV)$
Odifff_cus_down	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot (E_{\text{Odifff_cus_cus}}) \cdot eV)$
Odifff_cus_left	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot (E_{\text{Odifff_cus_bridge}}) \cdot eV)$
Odifff_cus_right	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot (E_{\text{Odifff_cus_bridge}}) \cdot eV)$
Odifff_cus_up	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot (E_{\text{Odifff_cus_cus}}) \cdot eV)$
React_bridge_down	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot E_{\text{react_Obridge_CObridge}} \cdot eV)$
React_bridge_left	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot E_{\text{react_Obridge_COcus}} \cdot eV)$
React_bridge_right	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot E_{\text{react_Obridge_COcus}} \cdot eV)$
React_bridge_up	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot E_{\text{react_Obridge_CObridge}} \cdot eV)$
React_cus_down	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot E_{\text{react_Ocus_COcus}} \cdot eV)$
React_cus_left	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot E_{\text{react_Ocus_CObridge}} \cdot eV)$
React_cus_right	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot E_{\text{react_Ocus_CObridge}} \cdot eV)$
React_cus_up	$(\beta \cdot h) \cdot (-1) \cdot \exp(-\beta \cdot E_{\text{react_Ocus_COcus}} \cdot eV)$
Species	
CO	
O	
empty	

Fig. 5: Processes can be added by point and click or by entering a chemical expression.

The conceptual parts of this topic guide predate the [kmcos 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 kmcos.

3.1 The Concept of Kinetic Monte Carlo

3.1.1 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

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

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 $(3^{100})^2 \approx 2.66 \cdot 10^{95}$ 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

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

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

3.1.2 Basic Kinetic Carlo Algorithm

- **Fix rate constants** k_{ij} initial state x_i , and initial time t
- while $t < t_{\max}$ do
 1. draw random numbers $R_1, R_2 \in]0, 1]$
 2. find l such that $\sum_{j=1}^l k_{ij} < k_{i,\text{tot}} R_1 < \sum_{j=1}^{l+1} k_{ij}$
 3. increment time $t \rightarrow t - \frac{\ln(R_2)}{k_{i,\text{tot}}}$
- end

3.1.3 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 distribution²

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

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

$$P(0, t_w) = e^{-k_{\text{tot}} t_w}, \quad (3.1)$$

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

$$\langle t_w \rangle = \frac{\int_0^\infty dt_w t_w e^{-k_{\text{tot}} t_w}}{\int_0^\infty dt_w e^{-k_{\text{tot}} t_w}} = \frac{1}{k_{\text{tot}}}.$$

Therefore at every step, we need to advance the time by a random number that is distributed according to (3.1). One can obtain such a random number from a uniformly distributed random number $R_2 \in]0, 1]$ via $-\ln(R_2)/k_{\text{tot}}$.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 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 Press, 3rd edition, ISBN:0521768589, p. 287. [link](#)

3.1.4 Further Reading

For a very practical introduction I recommend Arthur Voter's tutorial⁴ and Fichthorn⁵ 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 Reuter⁷.

3.2 Modelling Workflows

At the core of modelling lies the art to capture the most important features of a system and leave all others out. kmcos 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.

3.2.1 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 *kmcos 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 *kmcos view* in the export folder.

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](#)

3.2.2 kmcOS workflows

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

- *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.

- *kmcOS edit, the model GUI editor (deprecated)*

Open an existing project *.xml file with

```
kmcOS edit <project_name>.xml
```

and inspect or edit it through on screen

- *the CLI editor*

Open an existing project *.xml file with

```
kmcOS import <project_name>.xml
```

and edit the project interactively on the ipython console.

- *edit the XML file*

Just open the XML file of your *kmcOS* 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.

3.3 The kmcOS data model

The guide explains how *kmcOS* 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 *kmcOS* 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 ‘*kmcOS edit*’ (deprecated) with an xml file, *kmcOS* validates the XML file and stores the content in a *Project* instance. If you export source code, *kmcOS* 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

3.4 How the kmcos kMC algorithm works

kmcos 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 straightforward 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 kmcos 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 kmcos 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. kmcos 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 kmcos 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.

3.4.1 The kmcos $O(1)$ 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

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

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 greater detail in the next subsection.

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

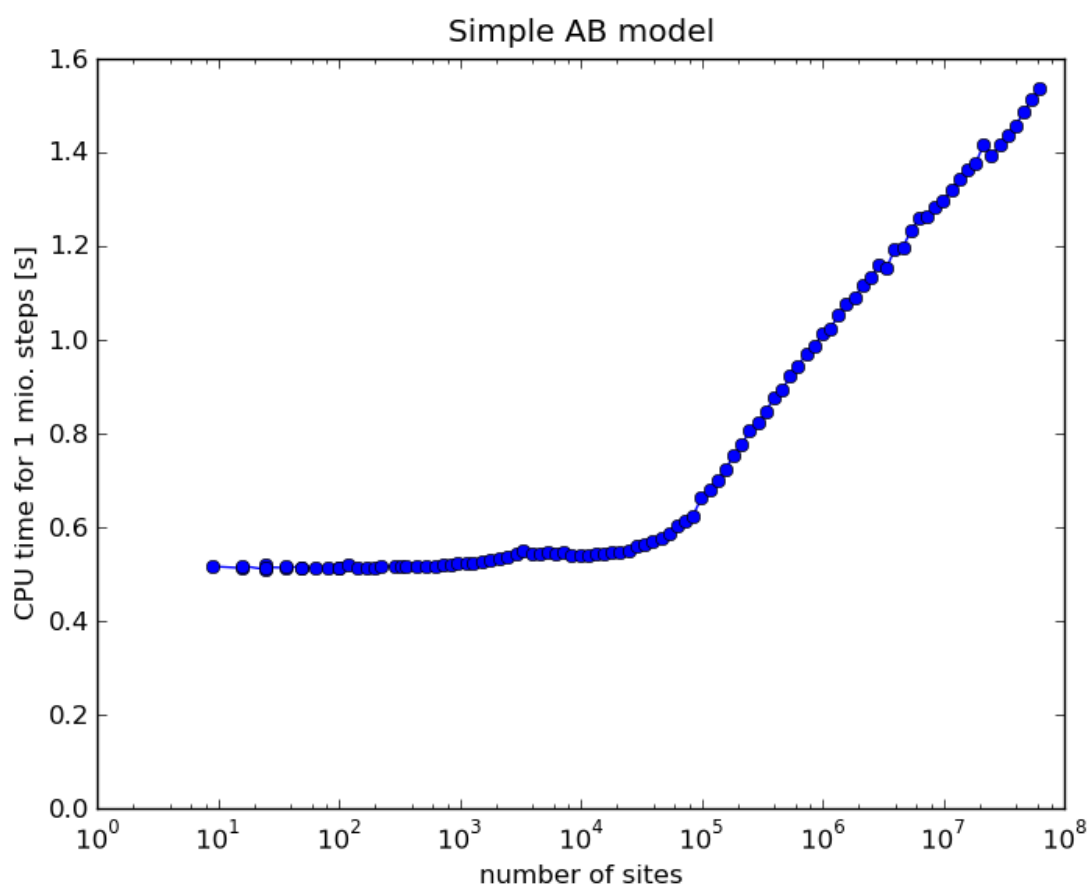


Fig. 1: Benchmark for a simple surface reaction model. All simulations have been performed on a single CPU of Intel I7-2600K with 3.40 GHz clock speed.

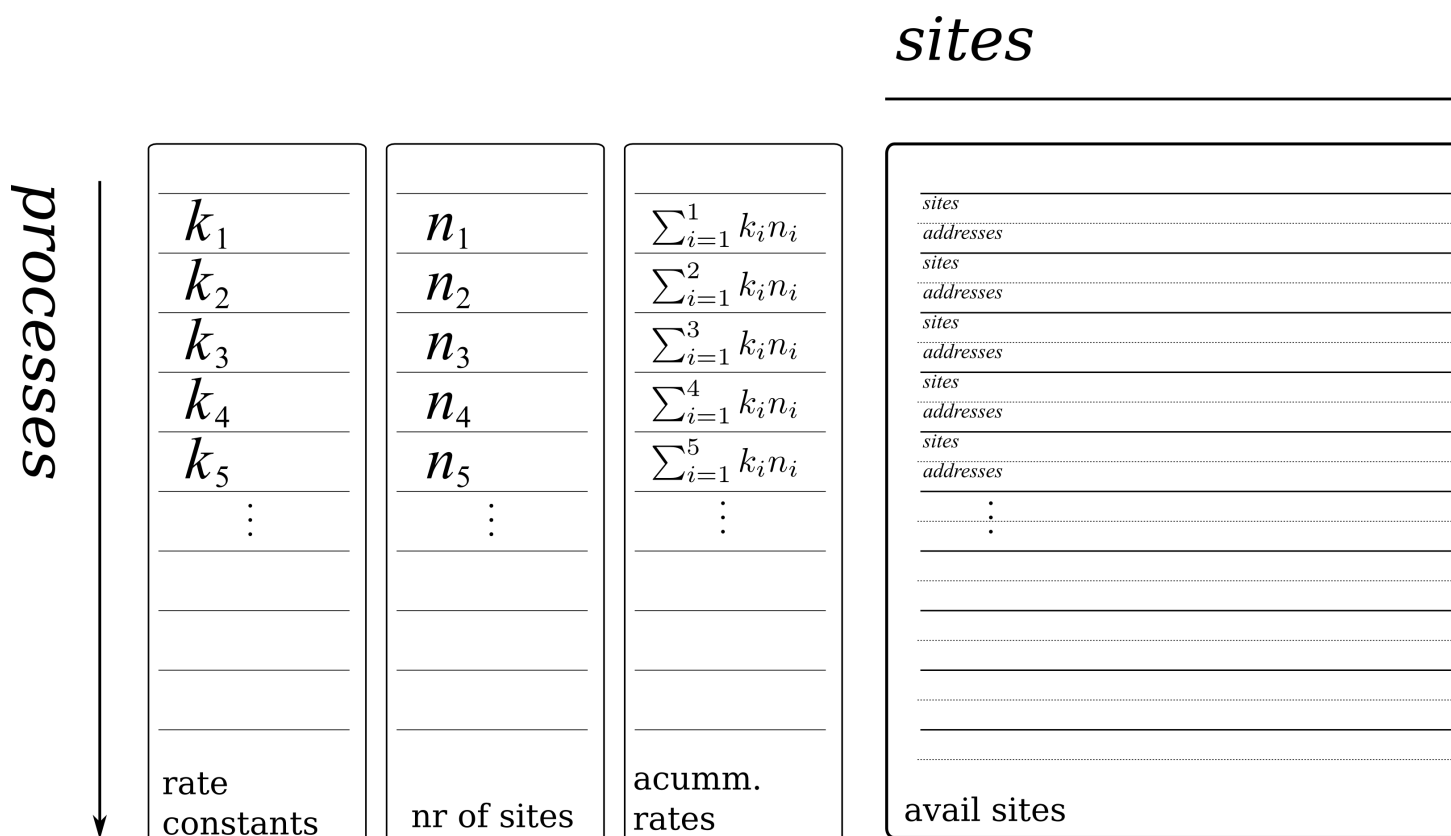


Fig. 2: The data model underlying the kmcos solver. The central component is the *avail_sites* array which stores for each elementary step the sites for which it is executable. Secondly it stores the location in memory, where the availability of the site is stored for direct access. The array of *rate 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.

- 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

3.5 Temporal acceleration

NOTICE: The temporal acceleration is still on an EXPERIMENTAL state. Please report any bugs encountered.

This implementation of a temporal acceleration algorithm attempts to deal with the low barrier problem often encountered in kinetic Monte Carlo simulations. It is based on the acceleration algorithm developed by Eric Christopher Dybeck, Craig Plaisance and Matthew Neurock, Journal of Chemical Theory and Computation, 13, 1525 (2017), <http://pubs.acs.org/doi/abs/10.1021/acs.jctc.6b00859>

The implementation in kmcos is published in: Mie Andersen, Craig Plaisance and Karsten Reuter Journal of Chemical Physics, 147, 152705 (2017), <http://aip.scitation.org/doi/full/10.1063/1.4989511>

In order for the scheme to work, it needs to be able to pair all processes into forward/reverse reactions. This is done according to the actions/conditions, where the forward process has the same actions as the conditions of the reverse process and vice versa. See the example model *render_co_oxidation_ruo2_processes_paired.py* from the examples folder for an example.

To enable acceleration, compile with the command: *kmcos export model.xml -t* or if using the backend for lateral interactions: *kmcos export model.xml -b lat_int -t*. Use syntax *kmos export model.xml -b otf -t* for the on the fly backend.

The model has four adjustable parameters (c.f. article):

Buffer_parameter (default: 1000): The smaller the value, the more aggressively the rate constants are scaled. Note that a good starting point is around the number of sites in the system.

Sampling_steps (default: 20): The number of kmc steps to take between each reassessment of the scaling factors. This parameter seems neither to be important for the accuracy nor the efficiency of the code. The default value should be fine.

Execution_steps (default: 200): The number of previous executions of either the forward or reverse process that is used to assess equilibrium. This parameter is also the number of executions of a forward/reverse process that must have occurred in the current superbasin for a process pair to be locally equilibrated. The default value seems to be close to the optimum efficiency for most systems tested so far.

Threshold_parameter (default: 0.2): This parameter is used to assess whether a given process pair is equilibrated. The efficiency of the algorithm worsens considerably if going below the default value of 0.2, whereas the accuracy of the algorithm is typically not too sensitive to the exact value.

Overall, the *buffer_parameter* seems to be the most important parameter for the accuracy of the algorithm, and one should always perform a careful convergence test with respect to this parameter before trusting the results. In the limit of an infinite value for the *buffer_parameter*, no scaling of the rate constants will be done.

It is possible to set these four parameters either when initiating a model: *model = KMC_Model(print_rates=False, banner=False, buffer_parameter=1000)* or one can use the set functions: *model.set_buffer_parameter(1000)*. Get

functions are also implemented: `model.get_buffer_parameter()`. Note that if you change the `execution_steps` after initializing the model, the model will be reset (as this parameter controls the length of some fortran arrays).

Accelerated kmc steps are run using the command: `model.do_acc_steps(nsteps)`.

In order to see what has happened during the simulations, one can use the commands `model.print_scaling_stats()` or `model.get_scaling_stats()`, which print/return the names of the paired processes as well as the average used scaling factors and the last set scaling factors (where the averages are typically higher (closer to 1), since scaling factors are reset to 1 every time a non-equilibrated reaction is carried out). Further implemented methods include `model.print_scaling_factors` and `model.print_proc_pair_eq`.

You can also use the command `model.set_debug_level(value: 0, 1 or 2)` to activate printing of certain fortran variables.

For models containing many different diffusion processes, the efficiency of the algorithm can be significantly increased by considering these processes to be equilibrated (but not locally equilibrated) by default. In practice this means that the execution of a diffusion process cannot cause the unscaling of all processes, as is normally the case whenever a non-equilibrated reaction occurs. However, diffusion processes still need to execute at least *execution_steps* times within the superbasin before being labeled as locally equilibrated and possible being subject to scaling. This above described behaviour is default for any process containing *diff* in the process name.

3.6 The otf Backend

NOTICE: The otf backend is still on an EXPERIMENTAL state and not ready for production.

As described in “*How the kmcOS kMC algorithm works*” the default kmcOS backends (`local_smart` and `lat_int`) produce code which executes in time $O(1)$ with the system size (total number of sites in the lattice). This is achieved through some book-keeping overhead, in particular storing every rate constant beforehand in an array. For some particular class of problems, *i.e.* those in which extended lateral interactions are taken into account. This implies that some elementary processes need to be included multiple times in the model definition (to account for the effect of the surrounding lattice configuration on the rate constants). Depending on the amount of sites taken in account and the number of different species that participate, the number of repetitions can easily reach several thousands or more. This leads to two undesired effects: First the amount of memory required by the book-keeping structures (which is proportional to the number of processes) could quickly be larger than your system has available. Second, the kmcOS algorithm is $O(1)$ in system size, but $O(N)$ in number of processes, which eventually leads to a slow down for more complex systems.

The otf backend was developed with these setbacks in mind. otf stands for On The Fly, because rate constants of processes affected by lateral interactions are calculated at runtime, according to user specifications.

NOTE: Up to now only a limited type of lateral interactions are supported at the moment, but the development of additional ones should be easy within the framework of the otf backend.

In this new backend, kmcOS is not able to generate $O(1)$ code in the system size, but now each process corresponds to a full group of processes from the traditional backends. For this reason, the otf backend is built to deal with simulations in which multisite/multispecies lateral interactions are included and in which the system size is not too large.

TODO: Put numbers to `when_to_use_otf(volume, nr_of_procs)`

3.6.1 Reference

Here we will detail how to set up a kmc model for the otf kmcOS backend. It will be assumed that the reader is familiar with Tutorial “*A first kMC Model—the API way*” and focus will be in the differences between the traditional backends (`local_smart` and `lat_int`) and otf. Most of the model elements (Project, ConditionAction, Species, Parameter) work exactly the same in the new backend.

The Process object, is the one whose usage is most distinct, as it can take two otf-backend exclusive attributes:

- `otf_rate`: Represent the expression to be used to calculate the rate of the process at runtime. It is parsed similarly to the `'rate_constant'` attribute and likewise can contain all the user defined parameters, as well as all constant and chemical potentials known to kmcos. Additionally, special keywords (namely `base_rate` and `nr_<species>_<flag>`) also have a special meaning. This is described below.
- `bystander_list`: A list of objects from the Bystander class (described below) to represent the sites which do not participate in the reaction but which affect the rate constant.

Additionally, the meaning of the `'rate_constant'` attribute is modified. This expression now represents the rate constant in the `'default'` configuration around the process. What this default configuration means is up to the user, but it will normally be the rate at the zero coverage limit (ZCL).

Additionally a new model description element, the Bystander, has been introduced. It has the attributes

- `coord`: Represents a coordinate relative to the coordinates in the process.
- `allowed_species`: This is a list of species, which can affect the rate constant of the process when they sit in `'coord'`
- `flag`: This is a short string that works as a descriptor of the bystander. It is useful when defining the `otf_rate` of the process to which the bystander is associated.

The rate constant to be calculated at runtime for each Process is given by the expression in `'otf_rate'`. Apart from all standard parameters, kmcos also parses the strings

- `'base_rate'`: Which is evaluated to the value of the `'rate_constant'` attribute

NOTE: For now, the `'base_rate'` expression is **required**.

- Any number of expressions of the form `'nr_<species>_<flag>'`. Where `<species>` is to be replaced by any of the species defined in the model and `<flag>` is to be replaced by one of the flags given to the bystanders of this process.

During export, kmcos will write routines that look at the occupation of each of the bystanders at runtime and count the total number of each species within `'allowed_species'` for each bystander type (flag).

3.6.2 Example

For this we will write down an alternative to the `render_pairwise_interaction.py` example file. Most of the script can be left as is. From the import statements, we can delete the line that imports `itertools`, as we won't be needing it. From then on, up to the point where we have defined all process not affected by lateral interactions, we do not need any changes. We also need to collect a set of all interacting coordinates which will affect CO desorption rate:

```
# fetch a lot of coordinates
coords = kmc_model.lattice.generate_coord_set(size=[2, 2, 2],
                                              layer_name='simplecubic_2d')
# fetch all nearest neighbor coordinates
nn_coords = [nn_coord for i, nn_coord in enumerate(coords)
              if 0 < (np.linalg.norm(nn_coord.pos - center.pos)) <= A]
```

as with traditional backends. With the otf backend however, we do not need to account for all possible combinations (and thus we do not need the `itertools` module). In this case, desorption only has one condition and one action:

```
conditions = [Condition(species='CO', coord=center)]
actions = [Action(species='empty', coord=center)]
```

And we use the coordinates we picked to generate some bystanders:


```
bystander_list = [Bystander(coord=coord,
                           allowed_species=['CO'],
                           flag='1nn') for coord in nn_coords]
```

As we are only considering the CO-CO interaction, we only include it in the allowed_species, but we could easily have included more species. Now, we need to describe the expressions to calculate the rate constant at runtime. In the original script, the rate is given by:

```
rate_constant = 'p_COgas*A*bar/sqrt(2*m_CO*umass/beta) '/
                '*exp(beta*(E_CO+%s*E_CO_nn-mu_COgas)*eV)' % N_CO
```

where the N_CO is calculated beforehand (in the model building step) for each of the individual lattice configurations. For the otf backend, we define the ‘base’ rate constant as the rate at ZCL (N_CO = 0), that is:

```
rate_constant = 'p_COgas*A*bar/sqrt(2*m_CO*umass/beta) '/
                '*exp(beta*(E_CO-mu_COgas)*eV) '
```

Finally, we must provide the expression given to calculate the rate given the amount of CO around in our bystanders. For this we simply define:

```
otf_rate = 'base_rate*exp(beta*nr_CO_1nn*E_CO_nn*eV) '
```

All of this comes together in the process definition:

```
proc = Process(name='CO_desorption',
              conditions=conditions,
              actions=actions,
              bystander_list = bystander_list,
              rate_constant=rate_constant,
              otf_rate=otf_rate)
kmc_model.add_process(proc)
```

3.6.3 Advanced OTF rate expressions

In the example above, the otf_rate variable for the processes included only a single expression that defined the rate taking into account the values of the nr_<species>_<flag> variables. For more complex lateral interaction models, this can become cumbersome. Alternatively, users can define otf_rate expressions that span several expressions/lines. Lets assume we are dealing with a model similar to the one above, but now include an additional species, O, and the corresponding lateral interaction energy E_CO_O between these two. Similarly to the previous example, the rate would be given by:

```
rate_constant = 'p_COgas*A*bar/sqrt(2*m_CO*umass/beta) '/
                '*exp(beta*(E_CO+%s*E_CO_nn+%s*E_CO_O-mu_COgas)*eV)' % (N_CO, N_O)
```

where N_O is the number of nearest-neighbour O. This rate expression is still fairly simple and the previously described method would work by doing:

```
otf_rate = 'base_rate*exp(beta*(nr_CO_1nn*E_CO_nn+nr_O_1nn*E_CO_O)*eV) '
```

However, equivalently (and maybe more easy to read) we could define:

```
otf_rate = 'vint = nr_CO_1nn*E_CO_nn+nr_O_1nn*E_CO_O\\n'
otf_rate += 'otf_rate = base_rate*exp(beta*vint*eV) '
```

in which we have defined an auxiliary variable `Vint`. Behind the scenes, these lines are included in the source code automatically generated by kmcOS. Notice the inclusion of explicit `\\n` characters. This is necessary because we want the line breaks to be explicitly stored as `\\n` in the `.xml` file for export (spaces are ignored by the xml export engine). Since these expressions are ultimately compiled as Fortran90 code, variable names are not case sensitive (i.e. `A = ...` and `a = ...` declare the same variable).

Additionally, when we want to include more than one line of code in `otf_rate`, we additionally need to include a line that states `otf_rate = ...` in order for kmcOS to know how to calculate the returned rate.

3.6.4 Running otf-kmcOS models

Once the otf model has been defined, the model can be run in a fashion very similar to the default kmcOS backends. Most of the differences arise from the

Todo: The rest of this sentence seems to have gotten lost somehow.

3.6.5 Known Issues

1. **Non-optimal updates to `rates_matrix`.** The current implementation of the backend is still non-optimal and can lead to considerable decrease in speed for larger systems sizes (scaling $O(N_{\text{sites}})$). This will be improved ($O(\log(N_{\text{sites}}))$) once more tests are conducted.
2. **Process name length limit** f2py will crash during compilation if a process has a name larger than approx. 20 characters.

3.7 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* corresponds 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 on 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.

3.7.1 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 kmcos.types import *
kmc_model = kmcos.create_kmc_model()

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

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

and the coordinate of a surface site

```
layer = Layer(name='default')
kmc_model.add_layer(layer)
layer.sites.append(Site(name='a'))
coord = kmc_model.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')])
kmc_model.add_process(adsorption)
```

Now this wasn't hard, was it?

3.7.2 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 = kmc_model.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')],
                       action_list=[Action(coord=coord,
                                           species='A')])
kmc_model.add_process(diffusion_up)
```

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

3.7.3 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. This may be incorrect, depending on how the rate constant value was defined. If the rate constant represents a net rate for adsorption / desorption from a pair of sites, (from an average or from a single transition state that bridges two sites), then the right,up,left, down procedure will have *double counted* the process because e.g. adsorption_up is the same processes as adsorption_down, just executed from one site above or below. One can compensate by dividing each adsorption rate constant by 2 and also each desorption rate constant by 2. One can also avoid double counting by only defining geometrically equivalent sites one time per unit cell: a simple trick is to only consider processes in the *positive* directions, for example.

However, note that if there are two transition state possibilities (one above one site, one above the other site) due to a heterolytic cleavage transition state, then there are supposed to be double processes and it is not double counting, just two paths that achieve the same reactants and products. Most rate constants in the literature for dissociative adsorption are defined as an average for the two sites, or involve a single homolytic transition state, and thus most cases of dissociative adsorption should have a single process between up and down as well as between left and right (yielding two processes, not four, for a simple cubic system).

3.7.4 Taking It Home

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

3.8 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 differentiation 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.

3.8.1 Manual generation

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

```
kmc_model.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
```

3.8.2 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 = kmc_model.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 you 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.

3.8.3 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

3.9 Developer's guide

3.9.1 Introduction and disclaimer

This guide intends to work as an introduction into kmcOS' internal structure, including both the automatically generated Fortran code, as well as the code generation procedure. As the name suggest, the intended audience are those who want to contribute to kmcOS as developers. This guide will assume that you are familiar with the way in which kmcOS is used. If that is not the case, you should start reading the sections of the [documentation](#) intended for users and/or go through the [Intro to kmcOS](#) tutorial.

DISCLAIMER: This information is provided with the hopes that it will ease your way into kmcOS' codebase, but it may contain errors. In the end only the interpretation of the code itself can really let you effectively add functionality to kmcOS.

For developers of the project, branch management will follow [this flowchart](#)

3.9.2 How to edit, install, and test your changes locally

First, install kmcOS, then you must locate your kmcOS installation. Typically it will be in a directory similar to: `~/VENV/kmcOS/lib/python3/site-packages/` If you have difficulty finding it, use `| python3 -c"import sys; print(sys.path)" |` Then, inside the site-packages directory, if kmcOS has an "egg" file, you must copy the kmcOS directory out of the egg file directly into the site_packages directory. After that, delete any other kmcOS egg files or directories so that there is a single kmcOS directory inside of site_packages.

OPTION 1 (recommended): Edit the source code directly inside site_packages, or edit the code elsewhere and then paste over the files in site_packages. Usually, this will be sufficient.

OPTION 2: Edit the source code elsewhere (such as in a shared folder) and reinstall using the setup.py:

```
source ~/VENV/kmcOS/bin/activate #this is the command to enter the python virtual_  
↪environment  
pip install .
```

This will reinstall kmcOS.

Before pushing to github, you should enter the tests directory and run the unit tests.

3.9.3 Some nomenclature

For some terms used frequently in this guide, there might exist some ambiguity on exact meaning. Here we present some definitions to try to alleviate this. Probably some ambiguity will remain, but hopefully not anything that cannot be discerned from context.

site: In kmcOS, this can have two different interpretations: either a specific node of the lattice or a type of site, e.g. a crystallographic site (top, fcc, hollow...). In this guide we use the former meaning: an specific position on the simulation lattice. When we need to indicate the second meaning, we will use **site type**.

coordinate: indicates the relative position of a site in the lattice with respect to some other site. In general, a coordinate will be given as a pair of site pair and an offset representing the relative positions of the unit cells (in units of the unit cell vector). The site used as reference will depend on the context.

process: a set of elementary changes that can occur on the lattice state on a single kMC step. A process is defined by a list of conditions and actions, and a rate constant expression.

executing coordinate: A coordinate associated to each process to be used as a reference for the relative positions of conditions and actions when defining the Fortran routines. In `local_smart` and `lat_int` the executing coordinate

is found with the help of the `kmcos.types.Process.executing_coord` method. In `otf` the concept of executing coordinate is not used, the reference position in the lattice is the central position implied by the user during model definition, i.e. the position in which coordinates have `offset = (0, 0, 0)`.

event: a process for which an specific site has been selected.

active event: An event that can be executed given the current state of the lattice, i.e. an event for which all associated conditions are fulfilled.

lateral interactions: For kmcos models built for the `local_smart` or `lat_int`, we will say that such model includes *lateral interactions* if there is one or more groups of processes with the following characteristics:

- a. their actions are all identical
- b. the conditions occurring on the same sites as the actions are identical
- c. there is a group of additional sites in which these processes have conditions, but these conditions are different for each process in the group

These processes represent the same change in the lattice, but under a difference state of the rest of the sites. These groups of processes are typically used to account for the effect of surrounding species on the values of the rate constants, i.e. the lateral interactions.

lateral interaction group: The group of processes defined by items a, b, c above.

bystander (`local_smart` or `lat_int` backends) The set of conditions of item c above, i.e. conditions of a process in a lateral interaction groups that do not have an action associated to it.

bystander site/coordinate: A site/coordinate associated with a bystanders.

participating sites: Sites associated with actions from item a and conditions from item b of the definition of a lateral interaction group.

lateral interaction event group: a collection of events occurring on the same lattice site and whose associated processes belong to the same lateral interaction group. Due to the nature of lateral interaction groups, only one of such events can be possible in a lattice at any given time.

bystander (`otf` backend): In the `otf` backend, the concept of bystander is explicitly included in the model definition, i.e. it is a new class `kmcos.types.Bystander` exclusive to this backend. An `otf` model is said to include lateral interactions if one of its processes includes such bystanders. Note that models *without* lateral interactions should not be built using the `otf` backend, as `local_smart` will definitely be more efficient.

3.9.4 The three *backends*

While most kmcos users will only need to worry about the Python interface to build and run the model, developers will also need to familiarize with the FORTRAN core code. The exact structure of this code depends on the *backend* that one selects. Which backend is most appropriate depends on the nature of the kMC model being implemented. Below we present a qualitative description of each backend.

`local_smart`

This is the original kmcos backend and has been used as a basis and inspiration for the rest of the backends. It was built with the implicit objective of offering the best run time performance at the expense of memory usage. For this reason, a key element in this backend is a precalculated list of rate constants, stored in the `base/rates` array.

This is the most efficient backend when the number of different rate constants list is reasonable small.

For models with very large number of different processes `nproc` (such as cases in which large lateral interaction groups exist) some undesirable effects can occur:

- The time needed to run a kMC step can become large, as it scales as $\mathcal{O}(n_{\text{proc}})$.
- The bookkeeping data structures, which scale in size with the total number of processes, can become too big for available memory.
- The size of individual source code files can become very large, making compilation very slow or even impossible due to memory requirements.

lat_int

The `lat_int` backend is the first attempt to alleviate the problems of `local_smart` for models with lateral interaction groups of moderate size. The main differences between them is that `lat_int` structures the generated code around the different lateral interaction groups and splits the source files accordingly. This way compilation is faster and requires less memory. A necessary consequence of this is that the logic for the lattice update needs to be different.

TODO: I seem to recall that there are models for which `lat_int` outperforms `local_smart`, even when `local_smart` can eventually compile and run. This should be verified and interpreted (i.e. is `lat_int` smarter than `local_smart` some times? If so, why?).

otf

For processes with lots of lateral interactions, i.e. very large lateral interaction groups, keeping a list of precalculated rate constants (and the proportionally large bookkeeping arrays) is unfeasible. The alternative is to evaluate rate constants during runtime, i.e. *on the fly*. kMC models built using the `otf` array do just that. To accommodate for this, the concept of a process in `otf` is different to that in the other backends. In `otf`, all members of a lateral interaction group are represented by a single process. Therefore, the total number of processes and, consequently, the size of bookkeeping arrays is much smaller. The counterpart from this improvement is that now a kMC step scales linearly with the system size (instead of being constant time).

3.9.5 The structure of the FORTRAN code.

Here we present a description of the different files in which the source code is split. We use the `local_smart` backend as a basis for this description, as it is the original backend and contains the fewest files. For the other backends, we will only explain the differences with `local_smart`.

All kmcOS models contain train main source files: `base.f90`, `lattice.f90` and `proclist.f90`. Each of these source files defines a module of the same name. These modules are exposed to Python interface.

It is important to know that some of the fortran code comes from the directory `kmcOSfortran_src` and some of the fortran code comes from the file `kmcOSio.py`, so `io.py` should be checked as well, if needed, when looking for fortran source code.

Files for the local_smart backend

base.f90

As its name suggests, `base.f90` contains the lowest-level elements of the model. It implements the kMC method in a 1D lattice. The `base` module contains all the bookkeeping arrays described in [Key data-structures](#) and the routines used to

- allocate and deallocate memory
- update of the bookkeeping arrays for lattice configuration and available processes
- using such arrays to determine the next process to be executed

- keep track of kMC time and total number of steps
- keep track of the number of executions of each individual process (`procstat`)
- saving an reloading the system's state

Many routines in `base` take a variable `site` as input. This is an index (integer value) that identifies a site on the 1D representation of the lattice (i.e. the ND lattice of the problem, flattened).

The contents of `base.f90` are (mostly) fixed, i.e. it is (almost) the same file for all kmcos models (as long as they use the same backend).

`lattice.f90`

The role of the `lattice.f90` is to generate the map from the ND lattice ($N=1, 2, 3$) to the 1D lattice that is handled by `base.f90`. The `lattice` module imports subroutines from the `base` module. Beside the look-up arrays `lattice2nr` and `nr2lattice`, used to map to and from the 1D lattice, this module also implements wrappers to many of the basic functions defined in `base.f90`. Such wrappers take now a 4D array `lsite` variable, designating the site on a 3D lattice, instead of the single integer `site` used by `base`. The first three elements of this array indicate the (x, y, z) position of the corresponding unit cell (in unit cell vector units), while the fourth indicates the site type. In cases of lower dimensional lattices, some elements of the `site` array simply stay always at a value of 0.

The `lattice.f90` file needs to be generated especially for each model, but only changes if the lattice used changes (e.g. if the number of site types or the dimension of the model).

`proclist.f90`

`proclist.f90` includes the routines called by the Python interface while running the model. In addition, it encodes the logic necessary to update the list of active events (i.e. the main bookkeeping arrays, `avail_procs` and `nr_of_sites`), given that a specific process has been selected for execution. The module imports methods and variables from both the `base` and `lattice` modules.

The `proclist.f90` files needs to be generated specially for each model, and is the file that changes most often during model development, as it is updated every time a process changes.

Files for the `lat_int` backend

`proclist.f90`

Some of the functionality that existed here in `local_smart` has been moved to different source files. While the functions called by the Python interface during execution remain here, the logic to update the list of active events is moved to `nli_*.f90` and `run_proc_*.f90` files. In addition, constants are also defined in an independent module on the separate file `proclist_constants.f90`.

`proclist_constants.f90`

Defines a module declaring several constants used by `proclist`, `nli_*` and `run_proc_*` modules.

`nli_<lat_int_nr>.f90`

There is one of such file for each lateral interaction group. These source files are enumerated starting from zero. Each of them implements a module called `nli_<lat_int_nr>` which contains a single function

`nli_<lat_int_group>`. `<lat_int_group>` is the name of the lateral interaction group, which coincides with the name of the first (lowest index) process in the group. These functions implement logic to decide which process from the group can occur on a given site, if any.

`run_proc_<lat_int_nr>.f90`

There is one of such file for each lateral interaction group. These source files are enumerated starting from zero. Each of them implements a module called `run_proc_<lat_int_nr>` that contains a single subroutine `run_proc_<lat_int_group>`. `<lat_int_group>` is the name of the lateral interaction group, which coincides with the name of the first (lowest index) process in the group. This routine is responsible of calling `lattice/add_proc` and `lattice/del_proc` for each lateral interaction group that should potentially be added or deleted. For this, it passes results of the `nli_<lat_int_group>` functions as argument, to ensure correct update of the list of active events.

Files for the `otf` backend

`proclist.f90`

Similar to `lat_int`, this file contains the functions called by the Python interface at runtime. Contrary to `local_smart`, the logic for the update of the active event list is in the `run_proc_<proc_nr>.f90` files and constants shared among different modules are defined on `proclist_constants.f90`.

`proclist_constants.f90`

Defines constant values to be shared between the `proclist`, `proclist_pars` and `run_proc_*`.

`proclist_pars.f90`

This file implements the modules `proclist_pars` (“process list parameters”) and takes care of providing functionality that only existed at the Python level in the earlier backends. More importantly, it implements the functions used to evaluate rate constants during execution. In more detail it:

- Implements the Fortran array `userpar` to access user-defined parameters at FORTRAN level, and functionality to update them from Python.
- When necessary, it implements a `chempots` array for accessing the chemical potentials in FORTRAN.
- It includes the routines `gr_<proc_name>` and `rate_<proc_name>`, which are used to evaluate the rate constants on the fly.

`run_proc_<proc_nr>.f90`

There is one of such file for each process in the model. They implement modules `run_proc_<proc_nr>` containing a `run_proc_<proc_name>` subroutine each. These routines contain the decision trees that figure out which events need to be activated or deactivated and call the corresponding functions from `base` (`add_proc` and `del_proc`).

3.9.6 Key data-structures

Here we describe the most important arrays required for bookkeeping in kmcOS. Understanding what information these arrays contain is crucial to understand how kmcOS selects the next kMC process to be executed. This is explained in *One kmc step in kmcOS*. All these data structures are declared in the `base` module and their dimensions are based on the “flattened” representation of the lattice in 1 dimension.

Important scalar variables

- `nr_of_proc` (int): The total number of processes in the model
- `volume` (int): The total number of sites in the lattice

Important arrays

`rates`

- Dimension: 1
- Type: float
- Size: `nr_of_proc`

Contains the rate constants for each process. This array is kept fixed during the execution of the kMC algorithm, and is only to be changed through the Python interface.

In the `otf` backend, rate constants are obtained on-the-fly during the execution of the kMC algorithm and stored in the `rates_matrix` array and the `rates` arrays contains simply a set of “default” rate constant values. These values can optionally (but not necessarily) be used to help with the calculation of the rates.

`lattice`

- Dimension: 1
- Type: int
- Size: `volume`

This array contains the state of the lattice, i.e. which species sits on each site.

`nr_of_sites`

- Dimensions: 1
- Type: int
- Size: `nr_of_proc`

This array keeps track of the number of currently active events associated to each process, i.e. it holds the number of different sites in which a given process can be executed.

accum_rates

- Dimensions: 1
- Type: float
- Size: nr_of_proc

This array is used to store partial sums of rate constants, ordered according to process index. In `local_smart` and `lat_int`, thanks to the fact that all copies of a process have an equal rate constant, the values of this array can be calculated according to

$$\text{accum_rates}(i) = \sum_{j=1}^i \text{rates}(j) * \text{nr_of_sites}(j) \quad (3.2)$$

In `otf` rate constants for a given process are different for a given site. Therefore, evaluation is more involved, namely

$$\text{accum_rates}(i) = \sum_{j=1}^i \sum_{k=1}^{\text{nr_of_sites}(j)} \text{rates_matrix}(j, k)$$

In all backends, the contents of `accum_rates` are reevaluated every kMC step.

avail_sites

- Dimensions: 3
- Type: int
- Size: nr_of_proc * volume * 2

This is arguably the most important bookkeeping array for `kmcos`, which keeps track of which processes can be executed each sites on the lattice, i.e. keeps track of all active events. To accelerate the update time of these arrays (see [here](#)), the information this array contains is duplicated. In practice, `avail_sites` can be considered as two 2D arrays of size `nr_of_proc * volume`.

Each row in `avail_sites(:, :, 1)` correspond to a process, and contains a list of the indices for the sites in which said process can occur according to the current state of the lattice, i.e. a list of the sites with active events associated to this process. Each site index appears at most once on each row. This array is filled from the right. This means that the first `nr_of_sites(i)` elements of row `i` will be larger than zero and smaller or equal than `volume`, while the last `(volume - nr_of_sites(i))` elements will all be equal to zero. The elements of the rows of `avail_sites(:, :, 1)` are **not** sorted, and their order depends on the (stochastic) trajectory the system has taken.

The rows on `avail_sites(:, :, 2)` function as an index for the rows of `avail_sites(:, :, 1)`. Given $1 \leq i \leq \text{nr_of_proc}$ and $1 \leq j \leq \text{volume}$, if process `i` can occur on site `j`, then `avail_sites(i, j, 2) = k`, with $k \geq 1$ and such that `avail_sites(i, k, 1) = j`. Conversely, if process `i` cannot occur on site `j`, then `avail_sites(i, j, 2) = 0` and no element in `avail_sites(i, :, 1)` will be equal to `j`.

procstat

- Dimensions: 1
- Type: long int
- Size Total number of processes (nr_of_proc)

```
avail_sites(. , . , 1)
```

3	7	2	9	0	0	0	0	0	0
5	4	1	10	2	7	9	8	0	0
6	3	5	2	4	8	0	0	0	0
2	1	8	0	0	0	0	0	0	0

```
avail_sites(. , . , 2)
```

0	3	1	0	0	0	2	0	4	0
3	5	0	2	1	0	6	8	7	4
0	4	2	5	3	1	0	6	0	0
2	1	0	0	0	0	0	3	0	0

Fig. 3: A example of an *avail_sites* array for a model with 5 processes and 10 sites.

This array is used to keep track of how many times each process is executed, i.e. the fundamental result of the kMC simulation. This array is used by the Python interface to evaluate the turnover frequencies (TOFs).

Additional arrays for the `otf` backend

The `otf` backend uses all the bookkeeping arrays from the other two backends, but needs in addition the following

`accum_rates_proc`

- Dimension: 1
- Type: float
- Size: volume

This array is updated in every kMC step with the accumulated rate for the process selected for execution. This is necessary because the site cannot be selected uniformly random from `avail_sites`, but needs to be picked with a binary search on this array.

`rates_matrix`

- Dimension: 2
- Type: float
- Size: `nr_of_proc * volume`

This matrix stores the rate for each current active event. The entries of this matrix are sorted in the same order as the elements of `avail_sites(:, :, 1)` and used to update the `accum_rates` array.

3.9.7 One kmc step in kmcos

The main role of the bookkeeping arrays from last section, specially `avail_sites` and `nr_of_sites`, is to make kMC steps execute fast and without the need to query the full lattice state. The routines `do_kmc_step` and `do_kmc_steps` from the `proclist` module execute such steps. The diagram above represents the functions called by these routines.

During system initialization, the current state of the system is written into the `lattice` array and the `avail_sites` and `nr_of_sites` arrays are initialized according to this. With these arrays in sync, it is possible to evaluate `accum_rates` according to eq. (3.2). With this information, and using two random numbers $0 < \text{ran_proc}, \text{ran_site} < 1$, the routine `base/determine_procsite` can select the next event to execute. This subroutine first selects a process according to the probabilities given by `accum_rates`. This is achieved by multiplying the total accumulated rate, i.e. the last element of `accum_rates`, times `ran_proc`. The subroutine `base/interval_search_real` implements a [binary search](#) to find the index `proc` such that

$$\begin{aligned} \text{accum_rates}(\text{proc} - 1) &\leq \\ \text{ran_proc} \cdot \text{accum_rates}(\text{nr_of_proc}) &\leq \\ \text{accum_rates}(\text{proc}). \end{aligned}$$

This step scales $O(\log(\text{nr_of_proc}))$. Then, a site is selected with uniform probability from the (non-zero) items of `avail_sites(proc, :, 1)`. This is valid because all individual events associated to a given processes share the same rate constant. This way, we avoid searching through the whole lattice, and we are able to select a `site` at constant time.

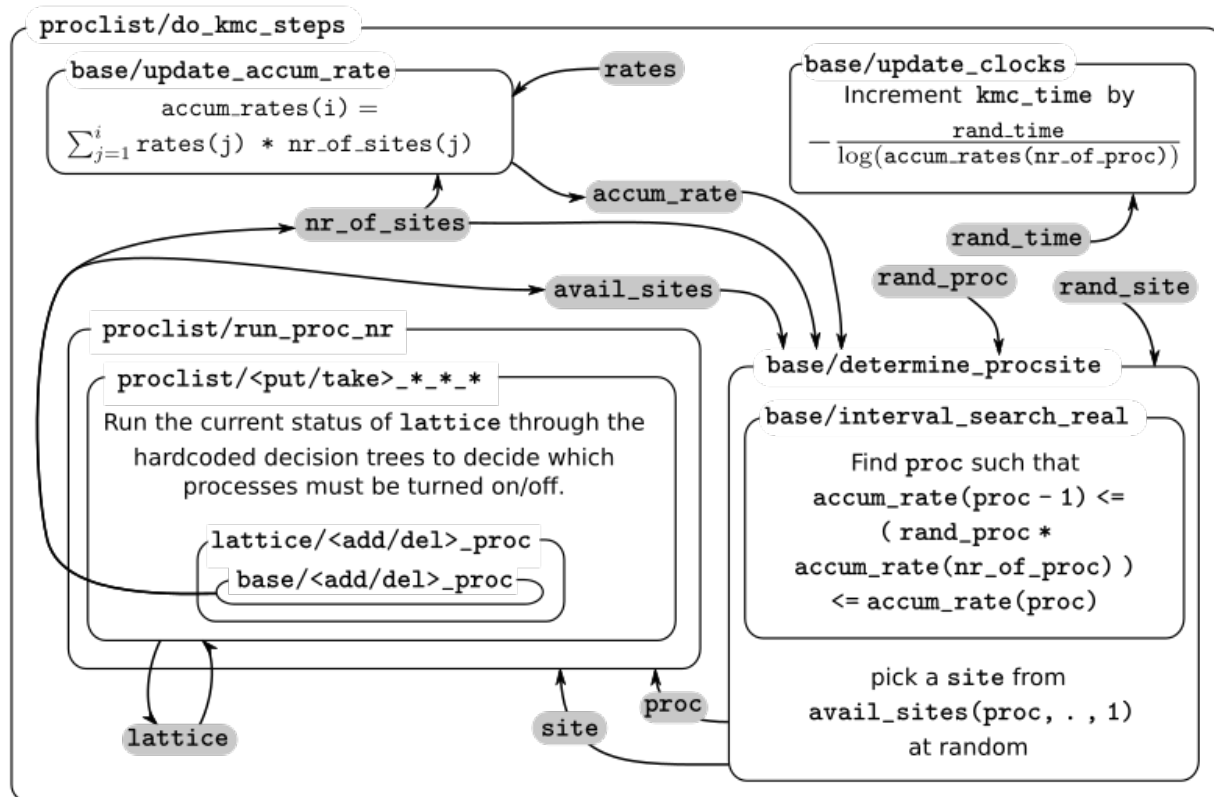


Fig. 4: A kMC step using kmcos' `local_smart` backend. Subroutines are represented by labeled boxes. The content of a given box summarizes the operations performed by the subroutine or the subroutines called by it. Variables (scalar or arrays) are indicated by gray boxes. An arrow pointing to a variable indicates that a subroutine updates it (or defines it). Arrows pointing to a subroutine indicate that the routine uses the variable. In kmcos, the passing of information occurs both through subroutine arguments and through module-wide shared variables; this distinction is not present in the diagram.

After this, the `proclist/run_proc_nr` subroutine is called with `proc` and `site` as arguments. This function first calls `base/increment_procstat` with `proc` as argument to keep track of the times each process is executed. Next, it uses the `nr2lattice` look-up table to transform the *scalar* `site` variable into the 4D representation (see [lattice.f90](#)). Finally, this function calls the methods which actually update the the lattice state and, consistent with this, the bookkeeping arrays. These are the `proclist/take_<species>_<layer>_<site>` and `proclist/put_<species>_<layer>_<site>` methods. Given a lattice site, take methods replace the corresponding species sitting there with the default species. The put methods do the converse. The set of put and take routines that need to be executed by each process are directly obtained from the conditions and actions from the process definition. These are hard-coded into the `proclist/run_proc_nr` routine, organized in a case-select block for the `proc` variable.

The `proclist/take_<species>_<layer>_<site>` and `proclist/put_<species>_<layer>_<site>` subroutines are arguably the most complex of a `local_smart` kmcOS model. Their ultimate goal is to call `lattice/add_proc` and/or `lattice/del_proc` to update `avail_sites` and `nr_of_sites` in correspondence with the change in the lattice they are effecting. To do this they need to query the current state of the lattice. The structure of these routines is described [below](#).

The actual update of `avail_sites` and `nr_of_proc` is done by the `base/add_proc` and `base/del_proc` functions. Under [Updating avail_sites](#) below, we explain how these functions make use of the structure of `avail_sites` to make updates take constant time. Once these arrays have been updated, the bookkeeping arrays are again in sync with the lattice state. Therefore, it is possible to reevaluate `accum_rates` using eq. (3.2) and start the process for the selection of the next step.

The put and take routines

These subroutines take care of updating the lattice and keeping the bookkeeping arrays in sync with it. When the occupation of a given site changes, some formerly active events need to be deactivated, while some formerly inactive events need to be activated. Figuring out which those events are is the main role of the put and take routines.

In kmcOS, processes are represented by a list of conditions and a list of actions. An event is active if and only if all the conditions of its associated process are satisfied. As the put and take routines only look at the change of an individual site in the lattice, determining which events need to be turned-off is straightforward: All active events which have a condition that gets unfulfilled on the site affected by the put/take routine will be deactivated. This is the first thing put/take routines do after updating the lattice.

Deciding which processes need to be activated is more involved. All inactive events with a condition that gets fulfilled by the effect of the put/take routine are candidates for activation. However, in this case, it is necessary to check the lattice state to find out whether or not such events have all other conditions fulfilled. A straightforward of accomplishing this is to sequentially look at each event, i.e.:

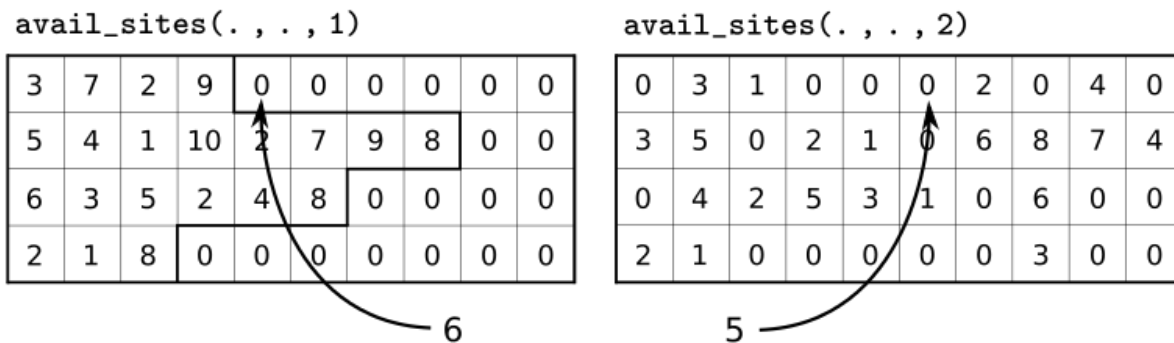
```
FOR each candidate event E
  TurnOn = True
  FOR each condition C of E
    IF C is unfulfilled:
      TurnOn = False
      break
  ENDIF
ENDFOR
IF TurnOn is True:
  Activate E
ENDIF
ENDFOR
```

However, chances are that many of the candidate events will have conditions on the same site. Therefore, a routine like the above would query a given lattice site many times for each execution of a put/take routine. For complex models with many conditions in the processes, this could become quickly the main computational bottleneck of the simulation.

The alternative to this naive approach, is to try to build a decision tree that queries the lattice state more efficiently. kmcOS generates such a decision tree using an heuristic algorithm. The main idea behind it is to group all the sites that would need to be queried and to sort them by the number of candidate events with conditions on them. A decision tree is built such that sites are queried on that order, thus prioritizing the sites that are more likely to reduce the number of processes that need activation. Such decision trees are implemented as select-case trees in the put/take routines and typically occupy the bulk of the code of `proclist.f90`. A more detailed description on how this is done is discussed [below](#).

Updating `avail_sites`

Adding `proc = 1`, `site = 6`



```
nr_of_sites(proc) ← nr_of_sites(proc)
avail_sites(proc, nr_of_sites(proc), 1) ← site
avail_sites(proc, site, 2) ← nr_of_sites(proc)
```

Fig. 5: Adding an process to the `avail_sites` array. Pseudocode for the addition of a process is also indicated.

The `avail_sites` and `nr_of_sites` arrays are only updated through the `base/add_proc` and `base/del_proc` subroutines, which take a process index `proc` and a site index `site` as input arguments. Adding events is programmatically easier. As the rows of `avail_sites(:, :, 1)` are filled from the left, the new event can be added by changing the first zero item of the corresponding row, i.e. `avail_sites(proc, nr_of_sites(proc) + 1, 1)`, to `site` and updating `avail_sites(:, :, 2)` and `nr_of_procs` accordingly. An example of this procedure is presented in the figure above.

Deleting an event is slightly more involved, as non-zero elements in `avail_sites(:, :, 1)` rows need to remain contiguous and on the left side of the array. To ensure this, the element that would be deleted (somewhere in the middle of the array) is updated to the value of the last non-zero element of the row, which is later deleted. To keep the arrays in sync, `avail_sites(:, :, 2)` is also updated, by updating the index of the moved site to reflect its new position. Finally, `avail_sites(site, proc, 2)` is set to zero. The figure above shows an example and presents pseudocode for such an update. Having the information in `avail_sites(:, :, 1)` duplicated (but restructures) in `avail_sites(:, :, 2)` allows these update operations to be performed in constant time, instead of needing to perform updates that scale with the system size.

Deleting proc = 3, site = 5

avail_sites(. , . , 1)

3	7	2	9	0	0	0	0	0	0
5	4	1	10	2	7	9	8	0	0
6	3	5	2	4	8	0	0	0	0
2	1	8	0	0	0	0	0	0	0

avail_sites(. , . , 2)

0	3	1	0	0	0	2	0	4	0
3	5	0	2	1	0	6	8	7	4
0	4	2	5	3	1	0	6	0	0
2	1	0	0	0	0	0	3	0	0

```

del_index = avail_sites(proc, site, 2)
move_site = avail_sites(proc, nr_of_sites(proc), 1)
avail_sites(proc, del_index, 1) ← move_site
avail_sites(proc, nr_of_sites(proc), 1) ← 0
avail_sites(proc, move_site, 2) ← del_index
avail_sites(proc, site, 2) ← 0
nr_of_sites(proc) ← nr_of_sites(proc) - 1

```

Fig. 6: Deleting an process from =avail_sites= array. Pseudocode for the deletion of a process is also indicated.

A kmc step with the lat_int backend

The process of executing a kMC step with the lat_int backend is very similar as that of the local_smart backend. In particular, the way avail_sites, nr_of_procs and accum_rates are updated, as well as the selection of process and site indices proc and site that will be executed is identical. The only difference exists within the call of the proclist/run_proc_nr routine, as the routines for finding which events need to be (de)activated are implemented differently.

In lat_int, proclist/proc_run_nr does not call put and take subroutines (which do not exist in the lat_int code-base), but calls subroutines specific to each lateral interaction group run_proc_<lat_int_nr>/run_proc_<lat_int_group>. They do not directly implement a decision tree, but rely on the nli_<lat_int_nr>/nli_<lat_int_group> functions.

The nli_<lat_int_nr>/nli_<lat_int_group> perform the analysis of the lattice state. They take a site on the lattice and look at the conditions of the elements of the corresponding lateral interaction event group. Using this information, they return the index of the process (within the lateral interaction group) which can currently be executed. If none can, it returns 0.

A proclist/run_proc_<lat_int_group> routine first calls del_proc for each lateral interaction event group which has a condition (including bystanders) affected by the changes in the lattice. The argument for del_proc will be the output of the corresponding nli_* functions, which will figure out which of the events is currently active (and can thus be deleted). After deleting processes, the lattice is updated according to the actions of the lateral interaction group. Once the new system state is set, add_proc is called for the same processes that del_proc was called, again using nli_* as argument. This way, the correct processes associated to the new state of the lattice will be activated.

This method works because of a slight, but important, difference in base/add_proc and base/del_proc between lat_int and local_smart. In local_smart, calling one of these functions with an argument proc=0

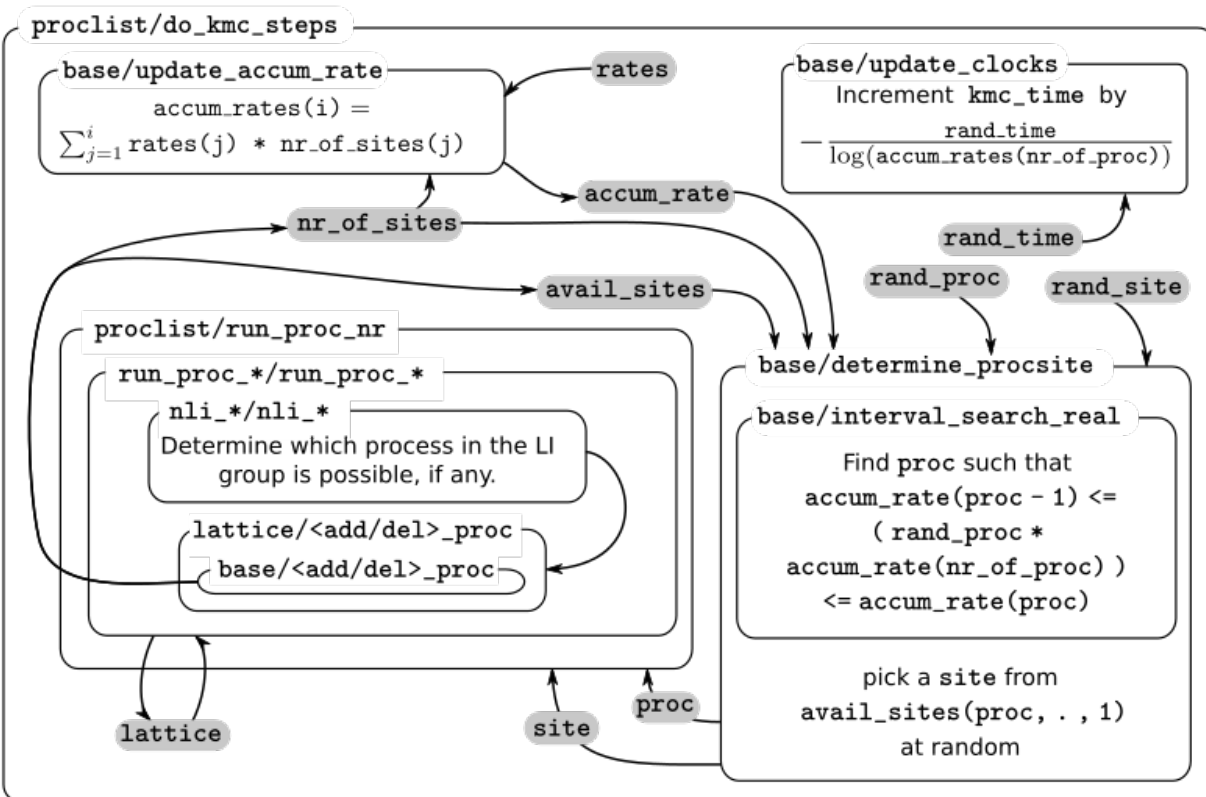


Fig. 7: A kMC step using `kmcos`' `lat_int` backend. Subroutines are represented by labeled boxes. The content of a given box summarizes the operations performed by the subroutine or the subroutines called by it. Variables (scalar or arrays) are indicated by gray boxes. An arrow pointing to a variable indicates that a subroutine updates it (or defines it). Arrows pointing to a subroutine indicate that the routine uses the variable. In `kmcos`, the passing of information occurs both through subroutine arguments and through module-wide shared variables; this distinction is not present in the diagram.

would lead to a program failure. In `lat_int`, this leads to the functions simply not adding or deleting any process to `avail_sites`.

A kmc step with the `otf` backend

As expected, the algorithm for running a kMC step with `otf` differs considerably from `local_smart` and `lat_int`. Firstly, the update of the `accum_rates` is more involved, as different copies of the processes do not share a single rate constant. For this reason, it is necessary to use the `rates_matrix` array, which contains the current rate constants for all active events. The `accum_rates` array is updated according to

$$\text{accum_rates}(i) = \sum_{j=1}^i \sum_{k=1}^{\text{nr_of_sites}(j)} \text{rates_matrix}(j, k)$$

The computational time to perform this summation now scales as $O(\text{nr_of_procs} \times \text{volume})$, instead of the $O(\text{nr_of_procs})$ for `local_smart`. Though this might seem like a disadvantage, it is important to notice that the value of `nr_of_procs` in `otf` can be smaller (potentially by several orders of magnitude) than in `local_smart`, and thus `otf` can outperform `local_small` for complex models (many lateral interactions) when using sufficiently small simulation sizes (small `volume`).

Once `accum_rates` is evaluated, `base/determine_procsite` proceeds to find the process index `proc` of the event to be executed. This is achieved by performing a binary search on `accum_rates`, exactly like in `local_smart` or `lat_int`. To select the site index, it is first necessary to evaluate

$$\text{accum_rates_proc}(i) = \sum_{k=1}^i \text{rates_matrix}(\text{proc}, k),$$

i.e. the partial sums of rates for the different events associated to process `proc`. Then a second binary search can be performed on `accum_rates_proc` to find `s` such that

$$\begin{aligned} \text{accum_rates_proc}(s-1) &\leq \\ \text{ran_site} - \text{accum_rates_proc}(\text{nr_of_sites}(\text{proc})) &\leq \\ \text{accum_rates_proc}(s). \end{aligned}$$

Therefore, `s` corresponds to the index of the selected site according to the current order of the `avail_sites(:, 1)` array. The site index as `site = avail_sites(proc, s, 1)`.

The process of updating the lattice and the bookkeeping arrays is also rather different. As in the other backends, first `proclist/run_proc_nr` is called with `proc` and `site` as arguments. Besides calling `base/increment_procstat`, it is responsible for calling the adequate `run_proc_<proc_nr>/run_proc_<proc_name>` routine. There is one of such routine for each process and they play the same role as the `put` and `take` routines in `local_smart`. The main difference is that these routines are built for executing full processes instead of elemental changes to individual sites. These functions need to look into the state of lattice and determine:

- a) which events get one or more of their conditions unfulfilled by the executed event
- b) which events get one or more of their condition fulfilled by the executed event and also have all other conditions fulfilled
- c) which events are affected by a change in one of their bystanders

For events in (a), `run_proc_<proc_nr>/run_proc_<proc_name>` run `lattice/del_proc`. For events in (b) and (c), rate constants are needed. This is done using functions from `proclist_pars` module, as described below. With the know rate constants, `run_proc_<proc_nr>/run_proc_<proc_name>` calls `lattice/add_proc` for each event in (b) and `lattice/update_rates_matrix` for each event in (c). In `otf`, `lattice/add_proc` and `base/add_proc` take a floating point argument for the rate constant in addition to

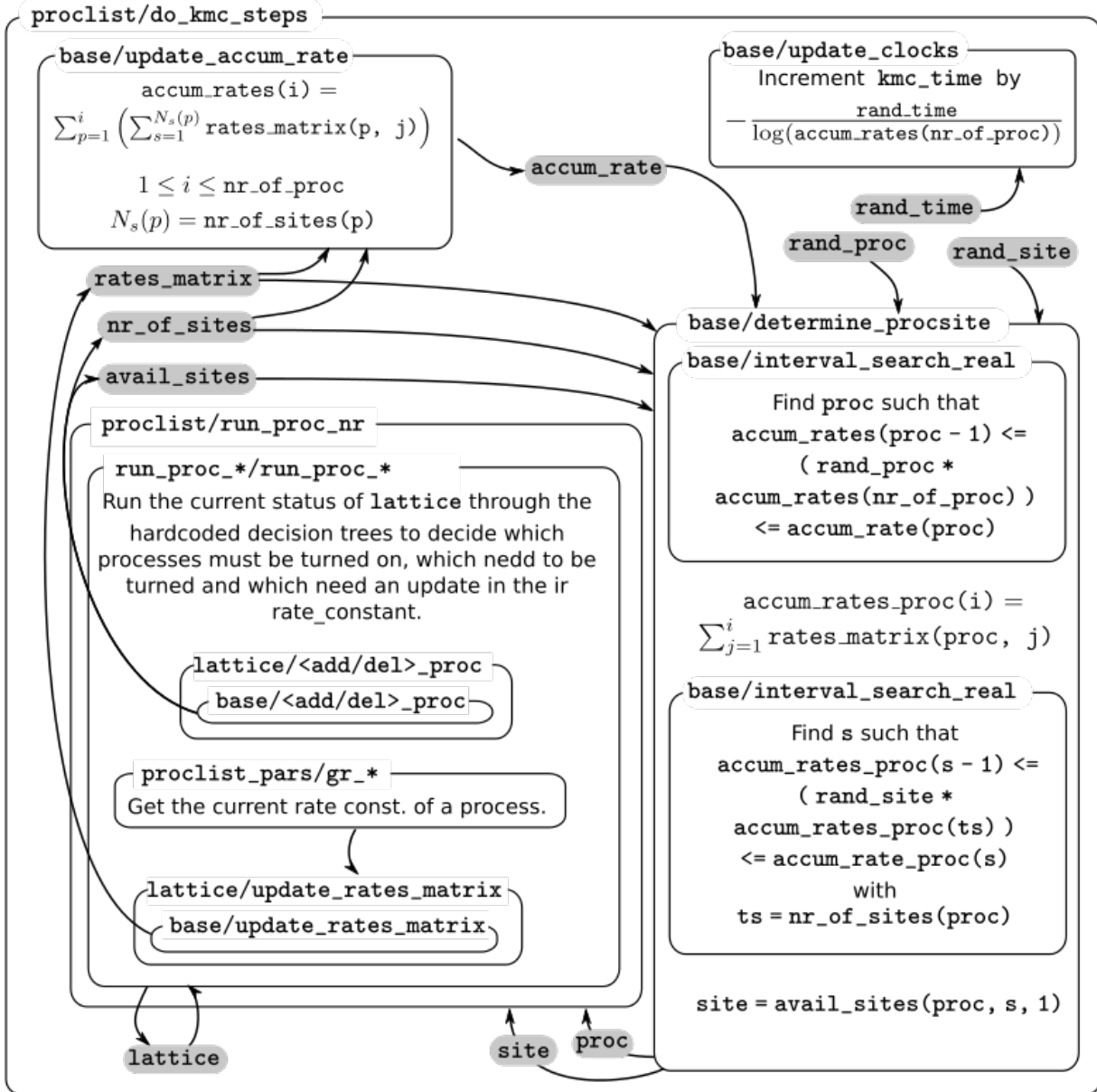


Fig. 8: A kMC step in with the `otf` backend. Subroutines are represented by labeled boxes, located inside the box corresponding to the calling function. Variables (scalar or arrays) are indicated by gray boxes. An arrow pointing to a variable indicates that a subroutine updates it (or defines it). An arrows pointing to a subroutine indicates that the routine uses the variable or the output of the function. The passing of information occurs both through subroutine arguments and through module-wide shared variables; this distinction is not present in the diagram.

the usual `site` and `proc` arguments. More details on the structure of these routines will be given in the section on the translation algorithm.

Rate constants are evaluated using the `proclist_params/gr_<proc_name>`. These functions look at the current state of the lattice to evaluate a integer array `nr_vars` which encodes the number of the different types of interactions that are present. This is used as input for the corresponding `proclist_pars/rate_<proc_name>` which implements the user defined rate expression. These can include user-defined parameters, which are encoded in FORTRAN with the `userpar` array in the `proclist_pars` module.

After `proclist/run_proc_nr` executes, the `lattice`, `avail_sites`, `nr_of_sites` and `rates_matrix` are in sync again, and the next kMC step can start with the evaluation of `accum_rates`.

3.9.8 The code generation routines

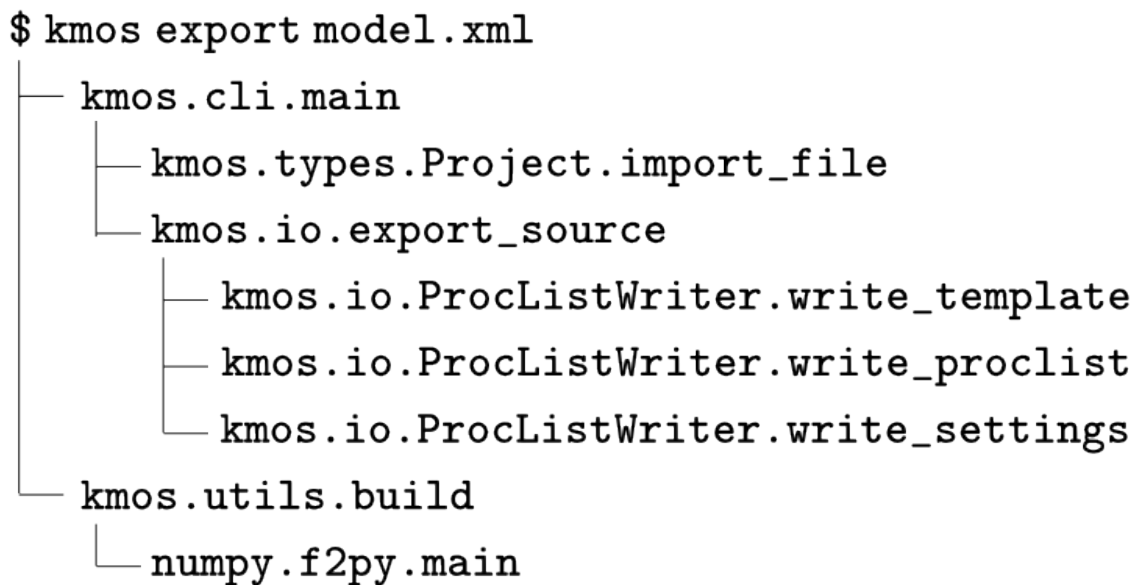


Fig. 9: Routines called during the export of a kmcOS model

As most of the source code described in the previous sections is generated automatically, it is crucial to also understand how this works. Code generation are contained in the `kmcOS.io` Python submodule. The normal way to use this module is through the command line, i.e. invoking the `kmcOS export` command. The figure [above](#) shows the subroutines/functions which are called when this is done. The command line call itself is handled by the `kmcOS.cli` submodule. Furthermore, the export procedure relies on the classes from the `kmcOS.types` submodule, which define the abstract representation of the kMC model. Specifically, a model definition from an `xml` or `ini` file into a `kmcOS.types.Project` object. The rest is done with the help of an instance of the `kmcOS.io.ProcListWriter` class, which contains several methods that write source code. Specifically, Fortran source code is generated in one of three ways:

- files are copied directly from kmcOS' installation
- code is generated with the help of a template file, which is processed by the `kmcOS.io.ProcListWriter.write_template` method
- code is written from scratch by one of the several `kmcOS.io.ProcListWriter.write_proclist_*` methods.

The format of the template files and how `kmcOS.io.ProcListWriter.write_template` works is explained in next section. The `kmcOS.io.ProcListWriter.write_proclist` method calls several other methods in charge of building different parts of the source code, these methods are named according to the pattern `kmcOS.io.ProcListWriter.write_proclist_*`. Exactly which of these methods are called depends on the backend being used. Some of such functions are specific to a certain backend, while other work for more than one backend. This is detailed under *The write_proclist method*.

The source file template

Template files are located in the `kmcOS/fortran_src/` folder of the `kmcOS`' source code and have the `.mpy` extension. Each line of these files contains either

- Python source code or
- template text prefixed with `#@`

`kmcOS.utils.evaluate_template` processes these files to convert them into valid python code. The Python lines are left unchanged, while the template lines are replaced by code adding the content of the line (i.e. things after the `#@`) to a string variable `result`. Template lines can contain placeholders, included as a variable name enclosed in curly brackets (`{` and `}`). If those variable names are found within the local variables of the corresponding `kmcOS.utils.evaluate_templates` call, the placeholders are replaced by the variable values. The `kmcOS.utils.evaluate_template` method accepts [arbitrary keyword arguments](#). In addition, the `kmcOS.io.ProcListWriter.write_template` is passed the current instance of the `ProcListWriter` class as `self`, the loaded kMC model information (i.e. the `kmcOS.types.Project`) instance as `data` and an options dictionary with additional settings as `options`.

With such template files it is possible to include some programmatically dependence on the model characteristics and other settings to an otherwise mostly static file. For example, in the `proclist_constants.mpy` template, the following text

```
for i, process in enumerate(self.data.process_list):
    ip1 = i + 1
    #@ integer(kind=iint), parameter, public :: {process.name} = {ip1}
```

is used to hard-coded the name constants used throughout the code to reference a process' index.

The write_proclist method

The scheme above shows the methods called by `kmcOS.io.ProcListWriter.write_proclist` to write `proclist.f90` and, for `lat_int` and `otf`, related files (`proclist_constants.f90`, `proclist_pars.f90`, `run_proc_*.f90`, `nli_*.f90`). All these `kmcOS.io.ProcList.write_proclist_*` methods take an `out` argument which is a [file object](#) to which the code is to be written and most take a `data` argument which is an instance of `kmcOS.types.Project` containing the abstract kMC model definition. Many of them also take a `code_generator` keyword argument with the backend's name. In what follows we briefly describe each of the individual methods. For clarity, they have been categorized according to the backend by which they are used. In cases in which the same routine is called to more than one backend, the description is presented only once.

Methods called to build local_smart source code

write_proclist_generic_part

This routine is only used by the `local_smart` backend. "Generic part" refers to the auxiliary constants defined in `proclist` (which exist in a separate file in `lat_int` and `otf`) and the functions whose code does not depend on the process details (e.g. `proclist/do_kmc_steps`).

```
ProcListWriter.write_proclist
├── if code_generator = 'local_smart'
│   ├── ProcListWriter.write_proclist_generic_part
│   │   ├── ProcListWriter.write_proclist_constants
│   │   └── ProcListWriter.write_proclist_generic_subroutines
│   ├── ProcListWriter.write_proclist_run_proc_nr_smart
│   ├── ProcListWriter.write_proclist_put_take
│   ├── ProcListWriter.write_proclist_touchup
│   ├── ProcListWriter.write_proclist_multilattice
│   └── ProcListWriter.write_proclist_end
├── if code_generator = 'lat_int'
│   ├── ProcListWriter.write_proclist_constants
│   ├── ProcListWriter.write_proclist_lat_int
│   │   ├── ProcListWriter._get_lat_int_groups
│   │   ├── ProcListWriter.write_proclist_lat_int_run_proc_nr
│   │   ├── ProcListWriter.write_proclist_lat_int_touchup
│   │   ├── ProcListWriter.write_proclist_generic_subroutines
│   │   ├── ProcListWriter.write_proclist_lat_int_run_proc
│   │   └── ProcListWriter.write_proclist_lat_int_nli_casetree
│   └── ProcListWriter.write_proclist_end
└── if code_generator = 'otf'
    ├── ProcListWriter.write_proclist_pars_otf
    ├── ProcListWriter.write_proclist_otf
    │   ├── ProcListWriter.write_proclist_generic_subroutines
    │   ├── ProcListWriter.write_proclist_touchup_otf
    │   ├── ProcListWriter.write_proclist_run_proc_nr_otf
    │   └── ProcListWriter.write_proclist_run_proc_name_otf
    └── ProcListWriter.write_proclist_end
```

Fig. 10: Routines used to write `proclist` and associated modules for the different backends.

write_proclist_constants

Uses the `proclist_constants.mpy` template to generate code defining named constants for the indices of each process and each species on the model. In `local_smart` this is added at the top of the `proclist.f90` file; in `lat_int` and `otf` this is included separately as the `proclist_constants.f90` file.

write_proclist_generic_subroutines

Uses the `proclist_generic_subroutines.mpy` template to write several routines not directly related with the tree search of process update, namely: `do_kmc_steps`, `do_kmc_step`, `get_next_kmc_step`, `get_occupation`, `init`, `initialize_state` and (only for `otf`) `recalculate_rates_matrix`.

write_proclist_run_proc_nr_smart

Writes the `proclist/run_proc_nr` function, which calls `put` and `take` routines according to the process selected by `base/determine_procsite`. This is basically a nested for-loop, first over the processes and then over the actions of such process. The only tricky part is to input correctly the relative coordinate for which the `take` and `put` routines need to be called. This is done with the help of the `kmcOS.types.Coord.radd_ff` method.

write_proclist_put_take

This is the most complex part of the `local_smart` code generator, in charge of writing a `put` and a `take` routine for each combination of site type and species in the model (except for the default species). These routines need to decide which events to activate or deactivate given an specific change in the lattice state.

The `write_proclist_put_take` is organized as several nested for loops. The outermost goes through each species in the model, the following through each layer and site type, and the next through the two possibilities, `put` and `take`. At this point, a specific `put_<species>_<layer>_<site>` or `take_<species>_<layer>_<site>` subroutine is being written.

For each of these routines, it is necessary to check which events (located relative to the affected site) can potentially be activated or deactivated by the operation being executed. This is done with further nested loops, going through each process and then through each condition of such process.

If a fulfilling match is found (i.e. the species and site type of the condition matches the site and species of a `put` routine or there is a condition associated to the default species on the site affected by a `take` routine) a *marker* to the corresponding process is stored in the `enabled_procs` list. This marker is a nested tuple with the following structure:

- first a list of `kmcOS.types.ConditionAction` objects (see below)
- then a tuple containing
 - the name of the process
 - the relative executing coordinate of the process with respect to the matching condition
 - a constant `True` value.

The list of `ConditionAction` objects contain an entry for each of the conditions of the given process, **except** for the condition that matched. The species are the same, but the coordinates of the these new `ConditionAction` objects are *relative* to the the coordinate of the matching condition. This way, we gain access to the position of the conditions of the events that can potentially be activated by the `put` or `take` routine relative to the position that is being affected in the surface. Note that potentially more than one marker could be added to the list for a given process. This would correspond to the possibility of different events associated to the same process being activated.

If an unfulfilling match is found, a tuple is added to the `disabled_procs` list. This tuple contains

- the process object and
- the relative position of the process with respect to the matching condition

There is less information in this case because the logic for disabling processes is much simpler than that for enabling them.

Once these `enabled_procs` and `disabled_procs` lists have been collected, a `del_proc` statement for each event in `disabled_procs` is written. Finally, the routine needs to write the decision tree to figure out which events to activate. This is done by the `kmcos.io.ProcListWriter._write_optimal_iftree` method, which calls itself recursively to build an optimized `select-case` tree.

`_write_optimal_iftree` expects an object with the same structure as the `enabled_procs` list as input. This is called `items` in the method's body. At the start, each entry of the list corresponds to an event that potentially needs to be activated. Associated to each of those, there is a list of all conditions *missing* for this events to be activated. If in the initial call to `_write_optimal_iftree` one of the events has no missing conditions (i.e. the corresponding list is empty), this means that their only condition was whatever the `put` or `take` routine provided. Consequently, the first step this method takes is to write a call to `add_proc` for those events (if any). Such events are then be removed from the `items` list.

Next the procedure that heuristically optimizes the if-tree starts. From `items`, it is possible to obtain the *most frequent coordinate*, i.e. that which appears most often within the lists of missing conditions. Such coordinate is selected to be queried first in the `select-case` tree. The possible cases correspond to the different possible species adsorbed at this coordinate. The routine iterates through those. For each species, it writes first the `case` statement. Then, the processes in `items` whose condition in the *most frequent coordinate* matches the current species are added to a reduced items list called `nested_items`. Next, the condition in the *most frequent coordinate* will be removed from the `nested_items`, creating the `pruned_items` list. This reduced list is used as input for a successive call to `_write_optimal_iftree`. The events that where included in `nested_items` are then removed from the `items` list.

It is possible (likely) that not all events will be have conditions in the most frequent coordinate. If this is the case, `_write_optimal_iftree` need to be called again to start an additional top-level case-tree to explore those processes.

In this way, further calls are made to `_write_optimal_iftree`, each of which in which the `items` list is shorter, of the item themselves contain fewer conditions. These calls “branch out”, but each branch eventually leads to calls with empty `items` list, which closes the corresponding branch. The decision tree finishes writing when all elements of `enabled_procs` have been exhausted.

`write_proclist_touchup`

This routine is in charge of writing the `proclist/touchup_<layer>_<site>`, one for each site type. These routines update the state of the lattice, one site at a time.

They first delete all possible events with executing coordinate in the current site. Then, they collect a list of all processes with executing coordinate matching the current site type. The list is built with the same structure as the `enabled_procs` list described in section (see [here](#)). This is then fed to the `_write_optimal_subtree` method, to build a decision tree that can decide which of those process are to be turned-on given the current state of the lattice.

`TODO write_proclist_multilattice`

`write_proclist_end`

This simply closes the proclist module with `end module proclist`.

Methods called to build `lat_int` source code

`write_proclist_lat_int`

This writes the header of the `proclist.f90` file for `lat_int` and then calls several `write_proclist_lat_int_*` functions in charge of writing the different routines of the module. Before it can do this, it needs to call `_get_lat_int_groups`, a method that finds all lateral interaction groups and returns them as a dictionary. This dictionary has the names of the groups as keys and the corresponding lists of processes as values. The name of a group is the name of the process within it with the lowest index (this coincides with the first process in the group when sorted alphabetically).

`write_proclist_lat_int_run_proc_nr`

This function is similar to its `local_smart` counterpart (see [here](#)). The only difference is that this routine needs to decide between lateral interaction groups instead of individual processes, as selecting the individual process within the group is done by the `nli_*` subroutines. For this reason, the indices of all processes of a group are included inside the `case(...)` statements.

`write_proclist_lat_int_touchup`

Writing the touchup functions is much simpler here than in `local_smart`, as here we can rely on the `nli_*` functions (see [here](#)). As in `local_smart`, all processes are deleted (just in case they were activated). Then `add_proc` is called for each lateral interaction group, using the result of the corresponding `nli_<lat_int_group>` function as input. Thus, an event will be added only if that function returns non-zero.

`write_proclist_lat_int_run_proc`

This method writes a `run_proc_<lat_int_nr>` module for each lateral interaction group. Each of these modules is located in its own file. The first step for writing the modules consists of finding all lateral interaction event groups which are affected by the actions of the current lateral interaction group. These are included in the list `modified_procs`. Once the list is built, a `del_proc` call is written for each of them, using the results of the corresponding `nli_<lat_int_group>` as argument. Then, it writes calls to `replace_species` to update the lattice. Finally a call to `add_proc` is added for each element of `modified_procs`, using the corresponding `nli_<lat_int_group>` as argument.

`write_proclist_lat_int_nli_casetree`

This method writes the `nli_*` routines, which decide which, if any, of the processes in a lateral interaction group can be executed in a given site of the lattice. For this, the method builds a nested dictionary, `case_tree`, which encodes the decision tree. This is then translated into a `select-case` Fortran block by the `kmcOS.io._casetree_dict` function.

Methods called to build `otf` source code

`write_proclist_pars_otf`

This method is only used by the `otf` backend. It is in charge of writing the `proclist_pars.f90` file. This module has two main roles: the first is to provide access to the user-defined parameters and other physical parameters and constants at the Fortran level. The second, to provide the routines which evaluate the rate constants during execution.

The routine first writes the declaration of the `userpar` array, used to store the value of the user-defined parameters. In addition, auxiliary integer constants (named as the parameters in the model) are declared to help with the indexing of this array. The `_otf_get_auxiliary_params` method is used to collect lists of constants, including the definitions of physical units, atomic masses and chemical potentials used in the rate expressions in the model. The constants and atomic masses are declared as constants with their corresponding value (evaluated using `kmcOS.evaluate_rate_expression`). If needed, a `chempot` array is included, which is used to store the value of the chemical potentials used in the model (auxiliary indexing variables are also included for this array).

In addition, this method writes a routine to update `userpar` from the Python interface, and another to read the values of such array. If needed, a routine to update `chempots` is also added.

In addition, this routine writes the functions used to evaluate the rate constants during execution. For each process, a `gr_<process_name>` and a `rate_<process_name>` are written. `gr_<process_name>` loops through all the bystanders to count how many neighbors of a given species there is for each “flag” associated to the process (see as determined by its `bystanders`). These counts are accumulated in the `nr_vars` array. This array is used as input to the corresponding `rate_<process_name>` routine. The content of this routine is directly obtained from the `otf_rate` attribute of the `kmcOS.types.Process` object. This user-defined string is processed by the `_parse_otf_rate` method to replace the standard parameter and constant names with the names understood by this Fortran module.

`write_proclist_touchup_otf`

This method writes the subroutines used to initialize the state of the bookkeeping arrays at the start of a simulation. For this, it calls the `_write_optimal_iftree_otf` with all possible events associated to the current site (i.e. with all processes). The routine `_write_optimal_iftree_otf` is very similar to the `_write_optimal_iftree` routine described used by `local_smart`’s `write_proclist_run_proc_nr_smart` (see [here](#)). The most remarkable difference is that in `otf` the `add_proc` routine needs to be called with the result of a `gr_<proc_name>` routine as an argument (to evaluate the current value of the event’s rate constant).

`write_proclist_run_proc_nr_otf`

The subroutine written by this method is very similar to its counterpart in the `lat_int` backend, only needing to decide which specific `run_proc_<procname>` function to call.

`write_proclist_run_proc_name_otf`

The `run_proc_<proc_name>` routines are the ones in charge of updating the bookkeeping arrays once a given event has been selected for execution. They are similar to their counterpart in `lat_int` in that there is one for each lateral interaction group. In `otf` there is only one process per “lateral interaction group”, so there is one such routine per process. They are also similar to the `put_*` and `take_*` subroutines from `local_smart` because they use very similar logic to build the hardcoded decision trees. The main difference between these backends is that the `run_proc_<proc_name>` routines of `otf` implement decision trees that take into account the changes in all sites affected by a process, while in `local_smart` `put_*` and `take_*` routines consider only an elementary change to a single site.

The first thing that `write_proclist_run_proc_name_otf` does is to collect a list with all the events for which one of the actions of the executing process unfulfills a condition (`inh_procs`), a list with all the processes for which they fulfill a condition (`enh_procs`) and a list with all the processes for which they modify the state of one of the bystanders (`aff_procs`). The processes that are included in `inh_procs` list are excluded from the other two lists.

Once this is done, calls to `del_proc` are written for all processes in `inh_procs`. Then, calls to the `replace_species` subroutine are added, so as to update the lattice according to the actions of the executing process. Afterwards, the subroutine `update_rates_matrix` is called for each process in `aff_procs` to update the corresponding rate constant.

As in the case of `local_smart` the most complex operation is that of activating processes, as the state of the lattice needs to be queried efficiently. To do this, a new list, `enabling_items`, is built based on the `enh_procs` list. `enabling_items` contains an entry for each process in `enh_process`. These entries are tuples containing:

- a list of conditions which are not satisfied by the executing event
- a tuple containing:
 - the name of the process
 - the relative position of the process with respect to the coordinate of the executing process
 - a constant `True` value.

This list is analogous to the `enabled_procs` list used by the `write_proclist_put_take` routine of the `local_smart` backend (see [here](#)). This list is used as input for the `_write_optimal_iftree_otf` method. This is very similar to the `_write_optimal_iftree`, with the only difference that calls to `add_proc` also need to include the result of the `gr_<proc_name>` functions as arguments.

4.1 Model running commands

4.1.1 Typical usage: `model.[command]`

Source code in: https://github.com/kmcos/kmcos/blob/master/kmcos/run/__init__.py

```
class kmcos.run.KMC_Model (image_queue=None, parameter_queue=None, signal_queue=None,
                             size=None, system_name='kmc_model', banner=True,
                             print_rates=False, autosend=True, steps_per_frame=50000, random_seed=None,
                             cache_file=None, buffer_parameter=None, threshold_parameter=None, sampling_steps=None,
                             execution_steps=None, save_limit=None)
```

API Front-end to initialize and run a kMC model using python bindings. Depending on the constructor call the model can be run either via directory calls or in a separate processes access via multiprocessing.Queues. Only one model instance can exist simultaneously per process.

`_adjust_database()`

Set the database of processes currently possible according to the current configuration.

`_get_configuration()`

Return current configuration of model.

Return type np.array

`_put (site, new_species, reduce=False)`

Works exactly like put, but without updating the database of available processes. This is faster for when one does a lot updates at once, however one must call `_adjust_database` afterwards.

Examples

```
# below puts a CO molecule at the `bridge` site of the lower left unit cell
model._put([0,0,0,model.lattice.bridge], model.proclist.co)
# below does the same:
model._put([0,0,0,"bridge"], "CO")
```

(continues on next page)

(continued from previous page)

```

model._put([1,0,0,model.lattice.bridge], model.proclist.co )
# below puts a CO molecule at the 'bridge' site one to the right

model._adjust_database() # Important !

```

Parameters

- **site** (*list or np.array*) – Site where to put the new species, i.e. [x, y, z, bridge]
- **new_species** (*str*) – Name of new species.
- **reduce** (*bool*) – Of periodic boundary conditions if site falls out site lattice (Default: False)

To see all the available site names, use `model.settings.site_names`, which come from `kmc_settings`.

Ex: `site_names = ['simple_cubic_hollow']` set site name as `'model.lattice.simple_cubic_hollow'`

Ex: `site_names = ['ruo2_bridge']` set site name as `'model.lattice.ruo2_bridge'`

To see all the available species names, one can use `model.settings.species_tags` which comes from `kmc_settings.py`.

Ex: `species_tags = { "CO":, "O":, "empty":, }`

`_set_configuration (config)`

Set the current lattice configuration.

Expects a 4-dimensional array, with dimensions [X, Y, Z, N] where X, Y, Z are the lattice size and N the number of sites in each unit cell.

Parameters `config` (*np.array*) – Configuration to set for model. Shape of array has to match with model size.

`create_configuration_plot (directory='./exported_configurations', plot_settings={}, showFigure=False, exportFigure=True, dimensionality=2)`

Returns the spatial view of the `kmc_model` and make a graph named `'plottedConfiguration.png'`, unless specified by `'figure_name'` in `plot_settings`

'coords' is expected to be the results from `get_species_coordinates(config, species, meshgrid = 'cartesian')`

Ex: `[[0 10 0] [0 11 0] [0 18 0] [1 6 0] [2 3 0] [2 11 0] [2 13 0]]` -> This is CO positions in [x y z]

`[[0 0 0] [0 1 0] [0 2 0] [0 3 0] [0 4 0] [0 5 0] [0 6 0]]` -> This is empty site positions in [x y z]

'directory' sets the directory name where the plot is saved

'plot_settings' is a dictionary that allows for the plot to change given the arguments

EX: `"y_label": "test", "x_label": "test", "legendLabel": "Species", "legendExport": False, "legend": True, "figure_name": "Plot", "dpi": 220, "speciesName": False`

'dimensionality' is an integer (either 2 or 3 dimensional) for the number of cartesian dimensions.

`deallocate ()`

Deallocate all arrays that are allocated by the Fortran module. This needs to be called whenever more than one simulation is started from one process.

Note that the current state and history of the system is lost after calling this method.

Note: explicit invocation was chosen over the `__del__` method because there seems to be an easy portable way to control garbage collection.

do_acc_steps (*n=10000, stats=True, save_exe=False, save_proc=0*)

Propagate the model *n* steps using the temporal acceleration scheme.

Parameters

- **n** (*int*) – Number of steps to run (Default: 10000)
- **stats** (*logical*) – Calculate statistics for the scaling factors
- **save_exe** (*logical*) – Track ‘save_limit’ number of executions following the execution of the target process ‘save_proc’
- **save_proc** (*integer*) – Process to be tracked

do_steps (*n=10000, progress=False*)

Propagate the model *n* steps.

Parameters **n** (*int*) – Number of steps to run (Default: 10000)

do_steps_time (*t=1.0, n=10000*)

Propagate the model *t* s, or *n* steps (whichever is achieved first. The *n* steps are intended to act as an upper limit to avoid infinite steps as well as for any other reason that the user may wish to limit the number of steps.

Parameters

- **t** (*real*) – Length of time (s) to run (Default: 1)
- **n** (*int*) – Upper limit for number of steps to run (Default: 10000)

Returns the number of iterations executed.

double ()

Double the size of the model in each direction and initialize larger model with current configuration in each copy.

dump_config (*filename, directory='./exported_configurations'*)

Use numpy mechanism to store current configuration in a file.

Parameters **filename** (*str*) – Name of file, to write configuration to.

export_movie (*filename="", directory='./exported_movies', resolution=150, scale=20, fps=1, frames=30, steps=1000000.0, representation='atomic', stitch=True*)

Exports a series of atomic view snapshots of model instance to a subdirectory, creating png files in the exported_movie_images directory and then creates a .webm video file of all the images of the images into a video

‘filename’ sets the filename for the images in the image directory and the video ‘scale’ increases the size of each species in the structure (currently not working as desired) ‘resolution’ changes the dpi of the images (currently not working as desired) ‘fps’ sets how long each image will stay in the video ‘frames’ sets the total video length ‘steps’ is the amount of steps the model does between each image

export_picture (*resolution=150, scale=20, filename="", **kwargs*)

Gets the atoms objects of the kmc_model and returns a atomic view of the configuration and make a file named ‘atomic_view.png’ unless specified by ‘filename’ in the function’s argument

‘filename’ sets the filename for the images in the image directory and the video

‘scale’ increases the size of each species in the structure (currently not working as desired)

‘resolution’ changes the dpi of the images (currently not working as desired)

get_atoms (*geometry=True, tag=None, reset_time_overrun=False*)

Return an ASE Atoms object with additional information such as coverage and Turn-over-frequencies attached.

The additional attributes are:

- *info* (extra tags assigned to species)
- *kmc_step*
- *kmc_time*
- *occupation*
- *proctat*
- *integ_rates*
- *tof_data*

tof_data contains previously defined TOFs in reaction per seconds per cell sampled since the last call to *get_atoms()*

info can be used to better visualize similar looking molecule during post-processing

proctat holds the number of times each process was executed since last *get_atoms()* call.

Parameters geometry (*bool*) – Return ASE object of current configuration (Default: True).

get_avail (*arg*)

Return available (enabled) processes or sites. If the argument is a sequence it is interpreted as a site (x, y, z, n). If it is an integer it is interpreted as a process.

param arg type or process to query

type arg int or [int]

get_backend ()

Return name of backend that model was compiled with.

Return type str

get_global_configuration (*filename_csv="", directory='./exported_configurations', export_csv=True, matrix_format='cartesian'*)

Gets each species and their respective coordinates and returns a 3d list that separates the coordinates of each species and EITHER returns a dictionary of the species's name OR returns a meshgrid of all the species

'filename_csv' sets the name of the csv file that will be exported if 'export_csv' is true

'directory' sets the directory name where the .csv file is saved if 'export_csv' is true

'export_csv' determines whether the functions exports the species coordinates as a csv file

'matrix_format' has two types of options: meshgrid and cartesian. Cartesian return as a csv with (x,y,z) format, and the meshgrid format returns as a csv with a XX, YY format

EX: Cartesian Species Coordinates CO [0,2,0] CO [0,4,0] CO [0,5,0] empty [0,0,0] empty [0,1,0] empty [0,3,0]

EX: Meshgrid [[0, 1, 1, 0, 1, 0], [1, 1, 1, 0, 1, 0], [0, 1, 1, 0, 1, 0], [1, 1, 0, 0, 1, 0], [1, 0, 1, 1, 1, 0], [1, 0, 1, 0, 1, 0]]

Note 1: For this case, "0" is empty and "1" is CO. In general, the meshgrid can have higher numbers representing more than 2 species if there are enough spaces in the model.

Note 2: The return value for `get_global_configuration()` and `get_species_coordinates()` have the same return values when setting `matrix_format = 'meshgrid'`

get_local_configurations (*configurationArray*, *radius=2*, *filename=""*, *directory='./exported_configurations'*, *export_files=True*, *unique_only=True*, *delimiter='|'*)

Takes in a meshgrid or `_config` object (from `_get_configuration`) and returns either the list of either all possible smaller meshgrids (i.e. the local configurations), or only the unique local configurations. Currently, `get_local_configurations` is only compatible with 2D configurations.

'meshgrid' is a matrix with all the species

EX: Meshgrid `[[0, 1, 1, 0, 1, 0], [1, 1, 1, 0, 1, 0], [0, 1, 1, 0, 1, 0], [1, 1, 0, 0, 1, 0], [1, 0, 1, 1, 1, 0], [1, 0, 1, 0, 1, 0]]`

Meshgrid can be obtained by calling `self.get_global_configuration(matrix_format='meshgrid')`

'radius' is the distance from the central species to the adjacent species EX: Radius = 1 `[[0, 1, 1,], [1, 1, 1,], -> This is one of the local configurations of the meshgrid [0, 1, 1,]`

'filename' sets the filename for the array of local configurations as a `.npz` file

'directory' sets the directory name where the `.npz` file is saved

'export_file' is the argument that determines if the function will export the return value as a `.npz` file

'unique_only' is the argument that determines if the function returns all possible local configurations or the unique local configurations

Example of the function's return value EX: `[[[0, 1, 1,], [1, 0, 1,], [0, 0, 1,]],`

`[[0, 1, 1,], [1, 1, 1,], [0, 1, 1,]],`

`[[1, 1, 1,], [1, 0, 0,], [0, 1, 1,]]]`

Note that if a `config` object is passed in, rather than a meshgrid, then each element is a list rather than an integer. For example, the first row might be `[[[0,1,0], [1,1,0] ...]`. However, the function will still work. So one can use `model._get_configuration()` and pass the output of that into `get_local_configurations`. A `config` object from `model._get_configuration()` should not be confused with a global configuration from `model.get_global_configuration()`, they are two representations of the global configuration but are very different in format.

#TODO: We should have an optional argument for the `configurationArrayFormat` which can be "meshgrid" versus "_config" since `_config` is really an internal format. Then we can use the flag and the "try and except" will be a last resort in an else statement.

get_next_kmc_step()

Returns the next kmc step's process and which site it would occur on, without taking the step. The output looks like this:

```
(Process model.proclist.o2_adsorption_bridge_right (13), Site (10, 19, 0, 1) [↪ #781])
```

The process name and process number are shown.

For the site, the format is the unit cell position in cartesian x,y,z followed by the site type's index (in this example, it is 1). As noted in the `"_put()"` function, the site type indexing starts at 1 (not at zero). One can use `model.settings.site_names` to see the site names, which come from `kmc_settings`. So a value of (10, 19, 0, 1) would mean unit cell 10,19,0 with site type `model.settings.site_names[0]` due to the different indexing.

get_occupation_header()

Return the names of the fields returned by `self.get_atoms().occupation`. Useful for the header line of an ASCII output.

get_param_header()

Return the names of field return by `self.get_atoms().params`. Useful for the header line of an ASCII output.

get_species_coordinates (*filename_csv*=", *directory*='.exported_configurations', *export_csv*=True, *matrix_format*='cartesian')

Gets the species coordinates from config and EITHER returns a 3d array, where each sub array lists the coordinates for a single species on the surface OR returns a meshgrid of all the species

'filename_csv' sets the filename for the functions return value as a .csv file if 'export_csv' is true

'directory' sets the directory name where the .csv file is saved if 'export_csv' is true

'matrix_format' has two types of options: meshgrid and cartesian. Cartesian return as a csv where each row represents the coordinates for a single species, and the meshgrid format returns as a csv with a XX, YY format

EX: Cartesian Ex: `[[0 10 0] [0 11 0] [0 18 0] [1 6 0] [2 3 0] [2 11 0] [2 13 0]]` -> This is CO positions in [x y z]

`[[0 0 0] [0 1 0] [0 2 0] [0 3 0] [0 4 0] [0 5 0] [0 6 0]]` -> This is empty site positions in [x y z]

EX: Meshgrid `[[0, 1, 1, 0, 1, 0], [1, 1, 1, 0, 1, 0], [0, 1, 1, 0, 1, 0], [1, 1, 0, 0, 1, 0], [1, 0, 1, 1, 1, 0], [1, 0, 1, 0, 1, 0]]`

Note 1: For this case, "0" is empty and "1" is CO. In general, the meshgrid can have higher numbers representing more than 2 species if there are enough spaces in the model.

Note 2: The return value for `get_global_configuration()` and `get_species_coordinates()` have the same return values when setting `matrix_format = 'meshgrid'`

get_std_sampled_data (*samples*, *sample_size*, *tof_method*='integ', *output*='str', *show_progress*=False)

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

[parameters] [TOFs] [occupations] kmc_time kmc_step

Parameter `tof_method` allows to switch between two different methods for evaluating turn-over-frequencies. The default method *procstat* evaluates the *procstat* counter, i.e. simply the number of executed events in the simulated time interval. *integ* will evaluate the number of times the reaction *could* be evaluated in the simulated time interval based on the local configurations and the rate constant.

Credit for this latter method has to be given to Sebastian Matera for the idea and implementation.

In each case check carefully that the observable is sampled good enough!

Parameters

- **samples** – Number of batches to average coverages over.
- **sample_size** (*int*) – Number of kMC steps in total.
- **tof_method** (*str*) – Method of how to sample TOFs. Possible values are *procrates* or *integ*. While *procrates* only counts the processes actually executed, *integ* evaluates the configuration to estimate the actual rates. The latter can be several orders more efficient for very slow processes. Differences resulting from the two methods can be used as an estimate for the statistical error in samples.

get_tof_header()

Return the names of the fields returned by self.get_atoms().tof_data. Useful for the header line of an ASCII output.

halve()

Halve the size of the model and initialize each site in the new model with a species randomly drawn from the sites that are reduced onto one. It is necessary that the simulation size is even.

load_config (filename, directory= './exported_configurations')

Use numpy mechanism to load configuration from a file. User must ensure that size of stored configuration is correct.

Parameters **filename** (*str*) – Name of file, to write configuration to.

nr2site (*n*)

Accepts a site index and return the site in human readable coordinates.

Parameters **n** (*int*) – Index of site.

Return type *str*

peek (*args, **kwargs)

Creates a static image of the model in a popup window.

play_ascii_movie (frames=30, steps=1, site=0, delay=0.1, species=None, hexagonal=False)

Shows a series of model snapshots in the current terminal. 'frames' sets the total video length 'steps' is the number of steps the model does between each image 'site' is the site of interest to animate 'species' is a list of species to display (default is all species)

plot_configuration (filename="", directory= './exported_configurations', resolution=150, scale=20, representation='spatial', plot_settings={}, showFigure=False, exportFigure=True)

Either calls create_configuration_plot() to create the spatial representation of the model, or calls export_picture() to create the atomic representation of the model

'filename' sets the filename for the plot

'directory' sets the directory name where the plot is saved

'scale' increases the size of each species in the structure (currently not working as desired)

'resolution' changes the dpi of the images (currently not working as desired) Note: 'resolution' and 'scale' are strictly for the atomic view

'representation' is an optional argument for spatial and atomic view You should specify as 'atomic' to see the atomic view. Leaving representation empty returns spatial view by default.

'plot_settings' is a dictionary that allows for the plot to change given the arguments EX:

```
"y_label": "test", "x_label": "test", "legendLabel": "Species", "legendExport": False, "legend":
True, "figure_name": "Plot", "dpi": 220, "speciesName": False
```

post_mortem (steps=None, propagate=False, err_code=None)

Accepts an integer and generates a post-mortem report by running that many steps and returning which process would be executed next without executing it.

Parameters

- **steps** (*int*) – Number of steps to run before exit occurs (Default: None).
- **propagate** (*bool*) – Run this one more step, where error occurs (Default: False).
- **err_code** (*str*) – Error code generated by backend if project.meta.debug > 0 at compile time.

print_accum_rate_summation (*order*='-rate', *to_stdout*=True)

Shows rate individual processes contribute to the total rate

The optional argument *order* can be one of: name, rate, rate_constant, nrofsites. You precede each keyword with a '-', to show in decreasing order. Default: '-rate'. Possible values are rate, rate_constant, name, nrofsites .

print_adjustable_parameters (*match*=None, *to_stdout*=True)

Print those methods that are adjustable via the GUI.

Parameters *pattern* (*str*) – fname pattern to limit the parameters.

print_coverages (*to_stdout*=True)

Show coverages (per unit cell) for each species and site type for current configurations.

print_kmc_state (*to_stdout*=True)

Shows current kmc step and kmc time.

procstat_normalized (*match*=None)

Print an overview view process names along with the number of times it has been executed divided by the current rate constant times the kmc time.

Can help to find those processes which are kinetically hindered.

Parameters *match* (*str*) – fname pattern to filter matching parameter name.

procstat_pprint (*match*=None)

Print an overview view process names along with the number of times it has been executed.

Parameters *match* (*str*) – fname pattern to filter matching parameter name.

put (*site*, *new_species*, *reduce*=False)

Puts *new_species* at *site*. The *site* is given by 4-entry sequence like [x, y, z, n], where the first 3 entries define the unit cell from 0 to the number of unit cells in the respective direction. And *n* specifies the site within the unit cell.

The database of available processes will be updated automatically. For doing many put and a single update, see the `_put()` function.

Examples

```
# below puts a CO molecule at the `bridge` site of the lower left unit cell
model.put([0,0,0,model.lattice.bridge], model.proclist.co)
# below does the same:
model.put([0,0,0,"bridge"], "CO")

model.put([1,0,0,model.lattice.bridge], model.proclist.co )
# below puts a CO molecule at the `bridge` site one to the right
```

Parameters

- **site** (*list* or *np.array*) – Site where to put the new species, i.e. [x, y, z, bridge]
- **new_species** (*str*) – Name of new species.
- **reduce** (*bool*) – Of periodic boundary conditions if site falls out site lattice (Default: False)

To see all the available site names, use `model.settings.site_names`, which come from `kmc_settings`.

Ex: `site_names = ['simple_cubic_hollow']` set site name as `'model.lattice.simple_cubic_hollow'`

Ex: `site_names = ['ruo2_bridge']` set site name as `'model.lattice.ruo2_bridge'`

To see all the available species names, one can use `model.settings.species_tags` which comes from `kmc_settings.py`.

Ex: `species_tags = { "CO":, "O":, "empty":, }`

run()

Runs the model indefinitely. To control the simulations, model must have been initialized with proper Queues.

show()

Creates a static image of the model in a popup window (this is a duplicate command of 'peek' created for convenience).

show_ascii_picture (*site, species, hexagonal=False*)

Shows an ascii picture of the current configuration.

start()

Start child process

view (*scaleA=None*)

Start current model in live view mode.

xml()

Returns the XML representation that this model was created from.

Return type str

class `kmcOS.run.Model_Rate_Constants`

Holds all rate constants currently associated with the model. To inspect the expression and current settings of it you can just call it as a function with a (glob) pattern that matches the desired processes, e.g.

```
model.rate_constant('*ads*')
```

could print all rate constants for adsorption. Given of course that 'ads' is part of the process name. The just get the rate constant for one specific process you can use

```
model.rate_constant.by_name("<process name>")
```

To set rate constants manually use

```
model.rate_constants.set("<pattern>", <rate-constant (expr.)>)
```

__call__ (*pattern=None, interactive=False, model=None*)

Return rate constants.

Parameters

- **pattern** (*str*) – fname pattern to filter matching parameter name.
- **model** (*kmcOS Model*) – runtime instance of kMC to extract rate constants from (optional)

by_name (*proc*)

Return rate constant currently set for *proc*

Parameters **proc** (*str*) – Name of process.

inverse (*interactive=False*)

Return inverse list of rate constants.

class `kmcOS.run.Model_Parameters` (*print_rates=True*)

Holds all user defined parameters of a model in concise form. All user defined parameters can be accessed and set as attributes, like so

```
model.parameters.<parameter> = X.Y
```

__call__ (*match=None, interactive=False*)

Return parameters that match ‘pattern’

Parameters **match** (*str*) – fname pattern to filter matching parameter name.

4.2 Connected Variables

The `connected_variables` dictionary allows a person to pass string-writable objects created during the model building into the runtime environment. This can be useful if a person needs access to some data structures (like lists of surrounding sites) during runtime. Dictionaries, strings, and lists can be passed. For more complex variables, one could pass the name of a pickle file. This feature is used for the `surroundingSitesDict`.

The basic syntax in a `build_file` would be as follows:

```
kmc_model = kmcOS.create_kmc_model(model_name)
kmc_model.connected_variables['frog_list'] = [1,2,3,4]
```

Then, during runtime, one could do the following:

```
print(model.connected_variables['frog_list'])
```

Additional Information for developers. Currently (Dec 2022), the way `kmcOS` processes things from the build file to the Runtime environment is as follows:

A person’s build file makes a Project class object (typically “`kmc_model`”), for example in https://github.com/kmcOS/kmcOS/blob/master/examples/MyFirstDiffusion__build.py That build file makes an xml file (or ini file), which occurs in `types.py` `_get_etree_xml` or `_get_etree_ini` where a string is made that then gets written to file. That xml/ini is then read back in and validated , which occurs against a DTD. A new Project class object is made from what is read back in. It is important to recognize that the new Project class object has many attributes that are the same as the one in the build file, but it is not the same object. It has fewer of the original attributes due to hardcoded mapping during xml writing and xml reading. When the source code compilation occurs, `kmc_settings`. is made. What is in `kmc_settings` roughly mirrors the original Project class object, but it is actually from the new Project class object that has been created from the xml.

4.3 Data Types

4.3.1 `kmcOS.types`

Holds all the data models used in `kmcOS`.

class `kmcOS.types.Project` (*model_name=None*)

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 Project class is primarily used in build files.

The overall structure is the following and was also displayed in the editor GUI (but that GUI is deprecated as of 2022).

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: $A@site1 + B@site2 \rightarrow empty@site1 + AB@site2$) 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 (**species*, ***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 kmcos.types.**Meta** (**args*, ***kwargs*)

Class holding the meta-information about the kMC project

class kmcos.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 kmcos.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', site_name=None*)

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

Parameters

- **name** (*str*) – Name of process.
- **rate_constant** (*str*) – Expression for rate constant.

- **otf_rate** (*str*) – Expression used to calculate rate on the fly using bystander’s configuration, off backend only!.
- **condition_list** (*list.*) – List of conditions (class Condition).
- **action_list** (*list.*) – List of conditions (class Action).
- **bystander_list** (*list.*) – List of bystanders (class Bystander), off backend only!.
- **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, ‘H2Ogas’: 2}. Hint: avoid space in keys.

class `kmcOS.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 `kmcOS.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.

4.3.2 kmcOS.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.

`kmcOS.io.export_source` (*project_tree*, *export_dir=None*, *code_generator=None*, *options=None*, *accelerated=False*)

Export a kmcOS 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.

The variable *project_tree* is a Project object (from *types.py*) and is analogous to the variable normally named *kmc_model* in a build file.

export_source is a central feature of the *kmcOS* approach. In order to generate different backend solvers, and allows additional candidates for kmc methods to be implemented.

`kmcos.io.export_xml` (*project_tree*, *filename=None*)

Writes a project to an XML file.

class `kmcos.io.ProcListWriter` (*data*, *dir*)

Write the different parts of Fortran 90 code needed to run a kMC model.

write_proclist (*smart=True*, *code_generator='local_smart'*, *accelerated=False*)

Write the proclist.f90 module, i.e. the rules which make up the kMC process list.

write_settings (*code_generator='lat_int'*, *accelerated=False*)

Write the kmc_settings.py. This contains all parameters, which can be changed on the fly and without recompilation of the Fortran 90 modules. In this function, “data” is an object analogous to what is normally in the variable “kmc_model” (within a build file), and it is a Project class object from types.py (just like kmc_model is). But this is not actually the same object, it is a fresh object made from the xml (or ini) file.

4.4 Editor frontend

4.4.1 kmcos.gui

4.4.2 kmcos.forms

4.5 Runtime frontend

4.5.1 kmcos.run

A front-end module to run a compiled kMC model. The actual model is imported in kmc_model.so and all parameters are stored in kmc_settings.py.

The model can be used directly like so:

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

model.parameters.T = 500
model.do_steps(100000)
model.view()
```

which, of course can also be part of a python script.

The model can also be run in a different process using the multiprocessing module. This mode is designed for use with a GUI so that the CPU intensive kMC integration can run at full throttle without impeding the front-end. Interaction with the model happens through Queues.

class `kmcos.run.ModelRunner`

Setup and initiate many runs in parallel over a regular grid of parameters. A standard type of script is given below.

To allow execution from multiple hosts connected to the same filesystem calculated points are blocked via <classname>.lock. To redo a calculation <classname>.dat and <classname>.lock should be moved out of the way

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

class ScanKinetics(ModelRunner):
```

(continues on next page)

(continued from previous page)

```

p_O2gas = PressureParameter(1)
T = TemperatureParameter(600)
p_COgas = PressureParameter(min=1, max=10, steps=40)
# ... other parameters to scan

ScanKinetics().run(init_steps=1e7, sample_steps=1e7, cores=4)

```

run (*init_steps=100000000.0, sample_steps=100000000.0, cores=4, samples=1, random_seed=None*)
Launch the ModelRunner instance. Creates a regular grid over all ModelParameters defined in the ModelRunner class.

Parameters **init_steps** – Steps to run model before sampling (i.e. to reach steady-state).

(Default: 1e8) :type init_steps: int :param sample_steps: Number of steps to sample over (Default: 1e8)
:type sample_steps: int :param cores: Number of parallel processes to launch. :type cores: int :param samples: Number of samples. Use more samples if precise coverages are needed (Default: 1). :type samples: int

class kmcOS.run.**ModelParameter** (*min, max=None, steps=1, type=None, unit=""*)

A model parameter to be scanned. If instantiated with only one value this parameter will be fixed at this value.

Use a subclass for specific type of grid.

Parameters

- **min** (*float*) – Minimum value for this parameter.
- **max** (*float*) – Maximum value for this parameter (Default: min)
- **steps** (*int*) – Number of steps between minimum and maximum.

class kmcOS.run.**PressureParameter** (**args, **kwargs*)

Create a grid of p in [p_min, p_max] such that ln({p}) is a regular grid.

class kmcOS.run.**TemperatureParameter** (**args, **kwargs*)

Create a grid of p in [T_min, T_max] such that ({T})⁻¹ is a regular grid.

class kmcOS.run.**LinearParameter** (**args, **kwargs*)

Create a regular grid between min and max.

class kmcOS.run.**LogParameter** (**args, **kwargs*)

Create a log grid between 10^{min} and 10^{max} (like np.logspace)

4.5.2 kmcOS.view

4.5.3 kmcOS.cli

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

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

```
kmcOS.cli.main('...')
```

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

```
kmcOS ex <xml-file>
```

instead of

```
kmcos export <xml-file>
```

etc.

You may also use syntax `kmcos.export("...")` for any cli command.

`kmcos.cli.main` (*args=None*)

The CLI main entry point function.

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

`$ kmcos <args>`

otherwise *args* will be taken from STDIN.

4.5.4 kmcos.utils

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

`kmcos.utils.build` (*options*)

Build binary with f2py binding from complete set of source file in the current directory.

`kmcos.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.

`kmcos.utils.get_ase_constructor` (*atoms*)

Return the ASE constructor string for *atoms*.

`kmcos.utils.split_sequence` (*seq, size*)

Take a list and a number *n* and return list divided into *n* sublists of roughly equal size.

`kmcos.utils.write_py` (*fileobj, images, **kwargs*)

Write a ASE atoms construction string for *images* into *fileobj*.

4.6 kmcos kMC project DTD

A standardized kmc model format has been made in XML. XML was chosen over JSON, pickle or alike because near 2010 it was the most flexible and universal format with good methods to define the overall structure of the data.

New infrastructure for JSON formats now exists, and it is on the to-do list to switch to using JSON to make a standard kmc model format.

One way to define an XML format is by using a document type description (DTD) and in fact at every import a kmcos 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
```

(continues on next page)

(continued from previous page)

```

    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

```

(continues on next page)

(continued from previous page)

```

    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
  >

```

4.7 Backends

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 *lattice* 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 *kmcos.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 kmcos.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](#).

4.7.1 local_smart

kmcos/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 `c` be the rate constants `n` the number of available sites, and `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):

- `system_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 **a** the integrated rates, **c** be the rate constants, **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 file named `<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 separated 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_step

Sets the current kmc_step

- kmc_step Writeable integer

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 identifier 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.

kmcOS/lattice

Implements the mappings between the real space lattice and the 1-D lattice, which kmcOS/base operates on. Furthermore replicates all geometry specific functions of kmcOS/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 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

lattice/default_layer

The layer in which the model is initially in by default (only relevant for multi-lattice models).

lattice/lattice2nr

Caching array holding the mapping from index to lattice coordinate: (x, y, z, n) -> i.

lattice/model_dimension

Store the number of dimensions of this model: 1, 2, or 3

lattice/nr2lattice

Caching array holding the mapping from index to lattice coordinate: i -> (x, y, z, n).

lattice/nr_of_layers

Constant storing the number of layers (for multi-lattice models > 1)

lattice/site_positions

The positions of (adsorption) site in the unit cell in fractional coordinates.

lattice/spuck

spuck = Sites Per Unit Cell Konstant The number of sites per unit cell, i.e. for coordinate notation (x, y, n) this is the maximum value of *n*.

lattice/system_size

Stores the current size of the allocated system lattice (x, y, z) in an integer array. In low-dimensional system, corresponding entries will be set to 1. Note that this should be thought of as a read-only variable. Changing its value at model runtime will not the indented effect of actually changing the simulated lattice. The definitive location for custom lattice size is *simulation_size* in *kmc_settings.py*.

If the system size shall be changed programmatically, it needs to happen before the *KMC_Model* is instantiated and Fortran array are allocated accordingly, like to

```
#!/usr/bin/env python3
import kmc_settings import kmcos.run
kmc_settings.simulation_size = 9, 9, 4
with kmcos.run.KMC_Model() as model: print(model.lattice.system_size)))‘
```

lattice/unit_cell_size

The dimensions of the unit cell (e.g. in Angstrom) of the unit cell.

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/do_kmc_steps_time

Performs a variable number of KMC steps to try to match the requested simulation time as closely as possible without going over. This routine always performs at least one KMC step before terminating. * Determine the time step for the next process * If the time limit is not exceeded, update clocks, rates, execute process,

etc.; otherwise, abort.

Ideally we would use *state(seed_size)* but that was not working, so hardcoded size.

t : Requested simulation time increment (input) *n* : Maximum number of steps to run (input) *num_iter* : the number of executed iterations (output)

proclist/get_next_kmc_step

Determines next step without executing it. However, it changes the position in the *random_number* sequence. The python function for *model.get_next_kmc_step()* should be used as it makes additional function calls to reset the random numbers. Calling *model.proclist.get_next_kmc_step()* is discouraged as that will call this subroutine directly and will not reset the random numbers.

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/get_seed

Function to retrieve the state of the random number generator to permit reproducible restart trajectories.

- 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/put_seed

Subroutine to set the state of the random number generator to permit reproducible restart trajectories.

- `state` an array of integers with the state of the random number generator (input)

proclist/run_proc_nr

Runs process `proc` on site `nr_site`.

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

proclist/seed_gen

Function to transform a single number into a full set of integers required for initializing the random number generator.

- `sd` an integer used to seed a simple random number generator
- used to generate additional integers for seeding the production random number generator (input)

kmcos/kind_values

This module offers `kind_values` for commonly used intrinsic types in a platform independent way.

4.7.2 lat_int

kmcos/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 c be the rate constants n the number of available sites, and 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):

- `system_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`, `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 integer, 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} be 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 or 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 $[0, \text{max_arr_value}]$

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 file named <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 separated 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 identifier 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.

kmcos/lattice

Implements the mappings between the real space lattice and the 1-D lattice, which kmcos/base operates on. Furthermore replicates all geometry specific functions of kmcos/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 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`

lattice/default_layer

The layer in which the model is initially in by default (only relevant for multi-lattice models).

lattice/lattice2nr

Caching array holding the mapping from index to lattice coordinate: $(x, y, z, n) \rightarrow i$.

lattice/model_dimension

Store the number of dimensions of this model: 1, 2, or 3

lattice/nr2lattice

Caching array holding the mapping from index to lattice coordinate: $i \rightarrow (x, y, z, n)$.

lattice/nr_of_layers

Constant storing the number of layers (for multi-lattice models > 1)

lattice/site_positions

The positions of (adsorption) site in the unit cell in fractional coordinates.

lattice/spuck

`spuck` = Sites Per Unit Cell Konstant The number of sites per unit cell, i.e. for coordinate notation (x, y, n) this is the maximum value of n .

lattice/system_size

Stores the current size of the allocated system lattice (x, y, z) in an integer array. In low-dimensional system, corresponding entries will be set to 1. Note that this should be thought of as a read-only variable. Changing its value at model runtime will not have the intended effect of actually changing the simulated lattice. The definitive location for custom lattice size is *simulation_size* in *kmc_settings.py*.

If the system size shall be changed programmatically, it needs to happen before the *KMC_Model* is instantiated and Fortran array are allocated accordingly, like to

```
#!/usr/bin/env python3
import kmc_settings import kmcos.run
kmc_settings.simulation_size = 9, 9, 4
with kmcos.run.KMC_Model() as model: print(model.lattice.system_size)))'
```

lattice/unit_cell_size

The dimensions of the unit cell (e.g. in Angstrom) of the unit cell.

kmcos/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/do_kmc_steps_time

Performs a variable number of KMC steps to try to match the requested simulation time as closely as possible without going over. This routine always performs at least one KMC step before terminating. * Determine the time step for the next process * If the time limit is not exceeded, update clocks, rates, execute process,

etc.; otherwise, abort.

Ideally we would use *state(seed_size)* but that was not working, so hardcoded size.

t : Requested simulation time increment (input) n : Maximum number of steps to run (input) num_iter : the number of executed iterations (output)

proclist/get_next_kmc_step

Determines next step without executing it. However, it changes the position in the random_number sequence. The python function for `model.get_next_kmc_step()` should be used as it makes additional function calls to reset the random numbers. Calling `model.proclist.get_next_kmc_step()` is discouraged as that will call this subroutine directly and will not reset the random numbers.

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/get_seed

Function to retrieve the state of the random number generator to permit reproducible restart trajectories.

- 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/put_seed

Subroutine to set the state of the random number generator to permit reproducible restart trajectories.

- `state` an array of integers with the state of the random number generator (input)

proclist/seed_gen

Function to transform a single number into a full set of integers required for initializing the random number generator.

- `sd` an integer used to seed a simple random number generator

used to generate additional integers for seeding the production random number generator (input)

kmcos/kind_values

This module offers `kind_values` for commonly used intrinsic types in a platform independent way.

4.7.3 otf

kmcos/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 up to a given process number taking into account all sites in which it is possible. ###

base/accum_rates_proc

Used to store the accumulated rate associated to each process ###

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):

- `system_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`, `proctat`.

`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 integer, 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 **a** the integrated rates, **c** be the rate constants, **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 or larger than 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 not 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 sight 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 currently possible process ordered as `avail_sites(:,1)`.

base/rates

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

base/reaccumulate_rates_matrix

Performs a process wide reaccumulation of the values in the `rates_matrix`. To be used when some of the user parameters are updated. Expected to alleviate some of the problems arising from floating point errors

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 file named `<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 separated 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 identifier 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/update_rates_matrix

Updates the `rates_matrix`. To be used when the state of a bystander has been modified

!

- `proc` positive integer number that represents the process whose rate is changed.
- `site` positive integer number that represents the site for the process

- `rate` positive real number that represents the updated rate

base/volume

Total number of sites.

base/walltime

Total CPU time spent on this simulation.

kmcos/lattice

Implements the mappings between the real space lattice and the 1-D lattice, which `kmcos/base` operates on. Furthermore replicates all geometry specific functions of `kmcos/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`

lattice/default_layer

The layer in which the model is initially in by default (only relevant for multi-lattice models).

lattice/lattice2nr

Caching array holding the mapping from index to lattice coordinate: (x, y, z, n) -> i.

lattice/model_dimension

Store the number of dimensions of this model: 1, 2, or 3

lattice/nr2lattice

Caching array holding the mapping from index to lattice coordinate: i -> (x, y, z, n).

lattice/nr_of_layers

Constant storing the number of layers (for multi-lattice models > 1)

lattice/site_positions

The positions of (adsorption) site in the unit cell in fractional coordinates.

lattice/spuck

spuck = Sites Per Unit Cell Konstant The number of sites per unit cell, i.e. for coordinate notation (x, y, n) this is the maximum value of *n*.

lattice/system_size

Stores the current size of the allocated system lattice (x, y, z) in an integer array. In low-dimensional system, corresponding entries will be set to 1. Note that this should be thought of as a read-only variable. Changing its value at model runtime will not the indented effect of actually changing the simulated lattice. The definitive location for custom lattice size is *simulation_size* in *kmc_settings.py*.

If the system size shall be changed programmatically, it needs to happen before the *KMC_Model* is instantiated and Fortran array are allocated accordingly, like to

```
#!/usr/bin/env python3
import kmc_settings import kmcos.run
kmc_settings.simulation_size = 9, 9, 4
with kmcos.run.KMC_Model() as model: print(model.lattice.system_size)))‘
```

lattice/unit_cell_size

The dimensions of the unit cell (e.g. in Angstrom) of the unit cell.

kmcos/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/do_kmc_steps_time

Performs a variable number of KMC steps to try to match the requested simulation time as closely as possible without going over. This routine always performs at least one KMC step before terminating. * Determine the time step for the next process * If the time limit is not exceeded, update clocks, rates, execute process,

etc.; otherwise, abort.

Ideally we would use `state(seed_size)` but that was not working, so hardcoded size.

t : Requested simulation time increment (input) *n* : Maximum number of steps to run (input) *num_iter* : the number of executed iterations (output)

proclist/get_next_kmc_step

Determines next step without executing it. However, it changes the position in the `random_number` sequence. The python function for `model.get_next_kmc_step()` should be used as it makes additional function calls to reset the random numbers. Calling `model.proclist.get_next_kmc_step()` is discouraged as that will call this subroutine directly and will not reset the random numbers.

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/get_seed

Function to retrieve the state of the random number generator to permit reproducible restart trajectories.

- 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/put_seed

Subroutine to set the state of the random number generator to permit reproducible restart trajectories.

- `state` an array of integers with the state of the random number generator (input)

proclist/run_proc_nr

Runs process `proc` on site `nr_site`.

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

proclist/seed_gen

Function to transform a single number into a full set of integers required for initializing the random number generator.

- `sd` an integer used to seed a simple random number generator
- used to generate additional integers for seeding the production random number generator (input)

kmcos/kind_values

This module offers `kind_values` for commonly used intrinsic types in a platform independent way.

4.8 Command Line Interface (CLI)

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

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

```
kmcos.cli.main('...')
```

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

```
kmcos ex <xml-file>
```

instead of

```
kmcos export <xml-file>
```

etc.

You may also use syntax `kmcos.export("...")` for any cli command.

4.8.1 List of commands

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

kmcOS 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.

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

kmcOS export <xml-file> [<export-path>] Take a kmcOS 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.

-t/--temp_acc
    Use temporal acceleration scheme.
    Builds the modules base_acc.f90, lattice_acc.mpy,
    proclist_constants_acc.mpy and
    proclist_generic_subroutines_acc.mpy.
    Default is false.

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

(continues on next page)

(continued from previous page)

```

--acf
    Build the modules base_acf.f90 and proclist_acf.f90. Default is false.
    This both modules contain functions to calculate ACF (autocorrelation_
    ↪function) and MSD (mean squared displacement).

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

```

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

kmcos help all Display documentation for all commands.

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

kmcos 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

kmcos run

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

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

kmcos shell

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

kmcos version Print version number and exit.

kmcos 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

kmcos xml Print xml representation of model to stdout

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 *kmcos.evaluate_rate_expression*. So you can try

```
from kmcos 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_NC1gas': {'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 kmcos is using internally? You can then print the explicit value for specific conditions in *kmcos shell*, for example like so

```
from kmcos 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. kmcos 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 *kmcos 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 kmcos 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 grained 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 *kmcos.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 kmcos 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

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)
- [Graph Theoretical KMC Code](#) (D. Vlachos)
- [Monty](#) (SXM Boerrigter)
- [MoCKa](#) (L. Kunz)
- [NASCAM](#) (S. Lucas)
- [Spparks](#) (S. Plimpton)
- [Zacros](#) (M. Stamatakis)

What is the relation between kmcos and kmos? Initially, the code was named kmos for *kinetic modeling on steroids*. However, during the 2020 revitalization and migration to python3, the code's name was changed kmcos with intention of greater emphasis of the generality of the code.

This document was generated Apr 12, 2023.

k

- `kmcos.cli`, 82
- `kmcos.io`, 80
- `kmcos.run`, 81
- `kmcos.types`, 76
- `kmcos.utils`, 83

Symbols

`__call__()` (*kmcos.run.Model_Parameters* method), 76
`__call__()` (*kmcos.run.Model_Rate_Constants* method), 75
`_adjust_database()` (*kmcos.run.KMC_Model* method), 67
`_get_configuration()` (*kmcos.run.KMC_Model* method), 67
`_put()` (*kmcos.run.KMC_Model* method), 67
`_set_configuration()` (*kmcos.run.KMC_Model* method), 68

A

`add_layer()` (*kmcos.types.Project* method), 77
`add_parameter()` (*kmcos.types.Project* method), 77
`add_process()` (*kmcos.types.Project* method), 77
`add_site()` (*kmcos.types.Project* method), 77
`add_species()` (*kmcos.types.Project* method), 78

B

`build()` (in module *kmcos.utils*), 83
`by_name()` (*kmcos.run.Model_Rate_Constants* method), 75

C

`ConditionAction` (class in *kmcos.types*), 80
`Coord` (class in *kmcos.types*), 80
`create_configuration_plot()` (*kmcos.run.KMC_Model* method), 68

D

`deallocate()` (*kmcos.run.KMC_Model* method), 68
`do_acc_steps()` (*kmcos.run.KMC_Model* method), 68
`do_steps()` (*kmcos.run.KMC_Model* method), 69
`do_steps_time()` (*kmcos.run.KMC_Model* method), 69
`double()` (*kmcos.run.KMC_Model* method), 69

`dump_config()` (*kmcos.run.KMC_Model* method), 69

E

`evaluate_kind_values()` (in module *kmcos.utils*), 83
`export_movie()` (*kmcos.run.KMC_Model* method), 69
`export_picture()` (*kmcos.run.KMC_Model* method), 69
`export_source()` (in module *kmcos.io*), 80
`export_xml()` (in module *kmcos.io*), 80

G

`generate_coord()` (*kmcos.types.LayerList* method), 79
`generate_coord_set()` (*kmcos.types.LayerList* method), 79
`get_ase_constructor()` (in module *kmcos.utils*), 83
`get_atoms()` (*kmcos.run.KMC_Model* method), 69
`get_avail()` (*kmcos.run.KMC_Model* method), 70
`get_backend()` (*kmcos.run.KMC_Model* method), 70
`get_global_configuration()` (*kmcos.run.KMC_Model* method), 70
`get_local_configurations()` (*kmcos.run.KMC_Model* method), 71
`get_next_kmc_step()` (*kmcos.run.KMC_Model* method), 71
`get_occupation_header()` (*kmcos.run.KMC_Model* method), 71
`get_param_header()` (*kmcos.run.KMC_Model* method), 72
`get_parameters()` (*kmcos.types.Project* method), 78
`get_processes()` (*kmcos.types.Project* method), 78
`get_species_coordinates()` (*kmcos.run.KMC_Model* method), 72
`get_speciess()` (*kmcos.types.Project* method), 78
`get_std_sampled_data()` (*kmcos.run.KMC_Model* method), 72

`get_tof_header()` (*kmcoss.run.KMC_Model method*), 72

H

`halve()` (*kmcoss.run.KMC_Model method*), 73

I

`import_xml_file()` (*kmcoss.types.Project method*), 78

`inverse()` (*kmcoss.run.Model_Rate_Constants method*), 75

K

`KMC_Model` (*class in kmcoss.run*), 67

`kmcoss` (*module*), 9

`kmcoss.cli` (*module*), 82

`kmcoss.io` (*module*), 80

`kmcoss.run` (*module*), 81

`kmcoss.types` (*module*), 76

`kmcoss.utils` (*module*), 83

L

`Layer` (*class in kmcoss.types*), 79

`LayerList` (*class in kmcoss.types*), 79

`LinearParameter` (*class in kmcoss.run*), 82

`load_config()` (*kmcoss.run.KMC_Model method*), 73

`LogParameter` (*class in kmcoss.run*), 82

M

`main()` (*in module kmcoss.cli*), 83

`Meta` (*class in kmcoss.types*), 78

`Model_Parameters` (*class in kmcoss.run*), 75

`Model_Rate_Constants` (*class in kmcoss.run*), 75

`ModelParameter` (*class in kmcoss.run*), 82

`ModelRunner` (*class in kmcoss.run*), 81

N

`nr2site()` (*kmcoss.run.KMC_Model method*), 73

P

`Parameter` (*class in kmcoss.types*), 78

`parse_and_add_process()` (*kmcoss.types.Project method*), 78

`parse_process()` (*kmcoss.types.Project method*), 78

`peek()` (*kmcoss.run.KMC_Model method*), 73

`play_ascii_movie()` (*kmcoss.run.KMC_Model method*), 73

`plot_configuration()` (*kmcoss.run.KMC_Model method*), 73

`pos` (*kmcoss.types.Coord attribute*), 80

`post_mortem()` (*kmcoss.run.KMC_Model method*), 73

`PressureParameter` (*class in kmcoss.run*), 82

`print_accum_rate_summation()` (*kmcoss.run.KMC_Model method*), 73

`print_adjustable_parameters()` (*kmcoss.run.KMC_Model method*), 74

`print_coverages()` (*kmcoss.run.KMC_Model method*), 74

`print_kmc_state()` (*kmcoss.run.KMC_Model method*), 74

`Process` (*class in kmcoss.types*), 79

`ProcListWriter` (*class in kmcoss.io*), 81

`procstat_normalized()` (*kmcoss.run.KMC_Model method*), 74

`procstat_pprint()` (*kmcoss.run.KMC_Model method*), 74

`Project` (*class in kmcoss.types*), 76

`put()` (*kmcoss.run.KMC_Model method*), 74

R

`run()` (*kmcoss.run.KMC_Model method*), 75

`run()` (*kmcoss.run.ModelRunner method*), 82

S

`show()` (*kmcoss.run.KMC_Model method*), 75

`show_ascii_picture()` (*kmcoss.run.KMC_Model method*), 75

`Site` (*class in kmcoss.types*), 79

`Species` (*class in kmcoss.types*), 79

`split_sequence()` (*in module kmcoss.utils*), 83

`start()` (*kmcoss.run.KMC_Model method*), 75

T

`TemperatureParameter` (*class in kmcoss.run*), 82

V

`validate_model()` (*kmcoss.types.Project method*), 78

`view()` (*kmcoss.run.KMC_Model method*), 75

W

`write_proclist()` (*kmcoss.io.ProcListWriter method*), 81

`write_py()` (*in module kmcoss.utils*), 83

`write_settings()` (*kmcoss.io.ProcListWriter method*), 81

X

`xml()` (*kmcoss.run.KMC_Model method*), 75