Scripting Manual

Amsterdam Modeling Suite 2025.1

www.scm.com

Mar 25, 2025

CONTENTS

CHAPTER

ONE

GETTING STARTED

1.1 Launching a terminal

To conveniently use the scripting tools of the Amsterdam Modeling Suite you need to set some shell environment variables and add the `AMSBIN` folder to your `PATH`. This can be done by sourcing file `amsbashrc.sh`, which is located in the Amsterdam Modeling Suite installation directory.

1.1.1 Linux

Note: if you followed the Linux Quickstart installation Guide, the `amsbashrc.sh` should be automatically sourced when you start up a new terminal, and you can ignore the following steps.

- Start up a terminal
- Source the `amsbashrc.sh` with the following command (note: you should replace `path_to_installation_directory` with the actual path to your AMS installation directory:

```
. path_to_installation_directory/amsbashrc.sh
```

- To test that you properly sourced the `amsbashrc.sh` file you can type the following command, which should yeld the help message for the *amsprep* (page 11) command line tool:

```
amsprep -h
```

1.1.2 MacOS

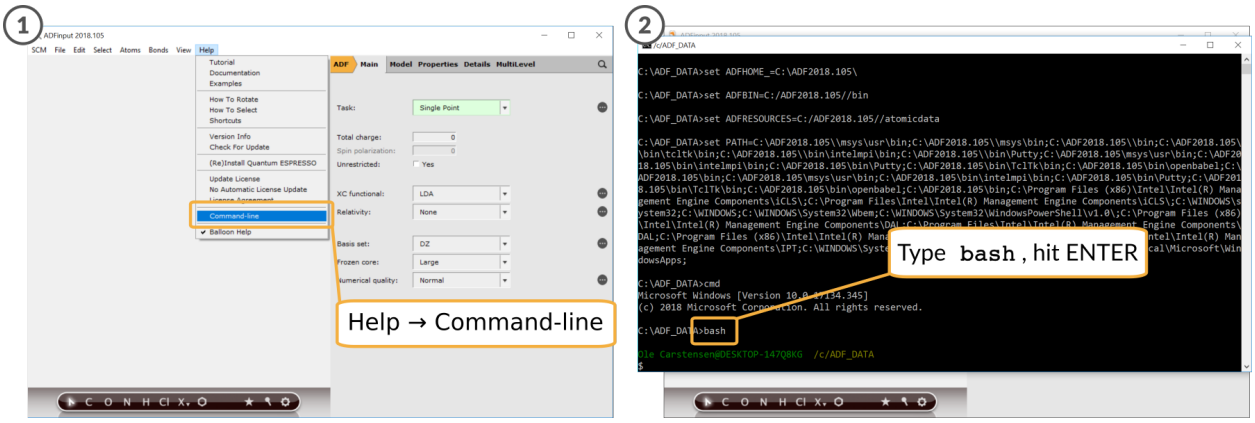
- From the AMSjobs GUI module, click on the **Help** dropdown menu and select **Terminal**. This will open a new terminal with all necessary environment variables already set and the `AMSBIN` folder already added to the `PATH`.

Alternatively, you can follow the Linux steps.

1.1.3 Windows

Every Windows installation of AMS2019 and newer, as well as older ADF versions, come with a pre-configured command line. The easiest way to access the command line is via the Help menu of the graphical user interface:

Go to **Help → Command-line**
Inside the command line window, type `bash` and hit ENTER (alternative: type `sh`)



The advantage of calling the command line from the GUI is that you will find yourself in the current working directory right away. In situations in which the GUI is not available, it is also possible to use the pre-configured command line directly:

Double click the file `ams_command_line.bat` in your AMS installation directory (e.g. C:/AMS2021.101)

1.2 Running amspython

Python scripts should be executed using the python3 interpreter shipped with Amsterdam Modeling Suite:

```
$AMSBIN/amspython scriptname.py
```

Further information can be found here: *amspython: Python Stack* (page 3).

CHAPTER

TWO

AMSPYTHON: PYTHON STACK

2.1 General

The Amsterdam Modeling Suite includes a Python stack based on the Enthought Python Distribution (https://www.enthought.com/products/epd/).

AMS version	Python version
AMS2023-AMS2024	3.8.12
AMS2020-AMS2022	3.6.9

This Python stack is completely separate from any other Python installations on the system.

All programs within the Amsterdam Modeling Suite launch Python via a special command, `amspython`. See *Getting Started* (page 1) and *Useful commands* (page 8).

2.2 Python packages available in amspython

The Python stack comes **pre-installed** with, for example:

- PLAMS: Python library for running and analyzing AMS calculations
- pyCRS: Python library for running and analyzing COSMO-RS calculations
- PISA : Python library for preparing AMS input files
- *SCM libbase* (page 33) : Python library for chemical structures, units, parsing AMS input files
- *AKFReader* (page 101): Python library for reading and parsing the AMS KF output files
- ASE: Use external calculators with AMS
- ASE: Use AMS as an ASE calculator
- pyZacros: Python library for Zacros Kinetic Monte Carlo simulations
- *FlexMD* (page 105): advanced Python-based molecular dynamics and QM/MM
- *AuToGraFS* (page 115): Python Library for MOFs, Zeolites, COFs, …
- RDKit
- Standard packages like numpy, scipy, matplotlib

Optional addon packages available through the AMS package manager:

- *Jupyter lab* (page 5),
- pandas,
- pymatgen,
- PyTorch,
- Tensorflow,
- and more

External community packages not developed by SCM:

- PyADF (https://github.com/chjacob-tubs/pyadf-releases) pursues the concept of automatizing the workflows required for multiscale simulations.
- tcutility (https://theochem-vu.github.io/TCutility/): Python library containing many helper functions and classes for use in programs written in the TheoCheM group

2.3 Installing new modules in the AMS Python Stack

1. **Package manager**

   Some Python packages, like pymatgen, pandas, and *Jupyter lab* (page 5), are installable through the AMS package manager.

2. **Pip**

   You can also extend the AMS Python Stack with by using `pip` to install additional modules if they are available on the Python Package Index (PyPI (https://pypi.python.org/pypi)). All `pip` commands need to be prefixed with `$AMSBIN/amspython -m`:

   ```
   $AMSBIN/amspython -m pip list
   $AMSBIN/amspython -m pip show scipy
   $AMSBIN/amspython -m pip search rotate-backups
   $AMSBIN/amspython -m pip install rotate-backups
   ```

   This will install the packages into a Python *virtual environment* (page 7).

   Warning: When you manually install packages into the AMS Python environment, you may break the SCM-supported ML Potential packages, for example by installing incompatible versions of dependencies. If this happens, it is easiest to remove the AMS Python *virtual environment* (page 7) completely and reinstall the ML Potential packages from the package manager.

3. **Modify SCM_PYTHONPATH**

   Alternatively, you can add the location of the source to the `SCM_PYTHONPATH` variable to make the module available in the AMS Python Stack. To avoid collisions with other Python installations on the system, we unload `PYTHONPATH` and `PYTHONHOME` from the environment when launching the ADF Python Stack and put the content of `SCM_PYTHONPATH` into `PYTHONPATH`.

   Hint: If you for some reason have to use the PYTHONPATH variable and are unable to use SCM_PYTHONPATH, you can modify $AMSBIN/amspython and $AMSBIN/amsipython to not have it cleared when starting python.

2.4 Install and run Jupyter Lab (Jupyter Notebooks)

2.4.1 Install Jupyter Lab

Install Jupyter lab using the Package Manager (GUI: SCM → Packages, Command-line: `$AMSBIN/amspackages gui` or `$AMSBIN/amspackages -h`).

2.4.2 Run Jupyter Lab

New in AMS2025: Select **SCM → Jupyterlab** in AMSjobs.

To launch Jupyter Lab from the command-line, run:

```
"$AMSBIN/amspython" -m jupyterlab
```

This will open up Jupyter Lab in your web browser.

To create a new notebook, click the **blue plus (+) button** at the top left and select **Notebook → Python 3 (ipykernel)**.

2.4.3 Run Jupyter notebooks through VS Code

Method 1: Use an existing Jupyter server in VS Code

To connect the VS Code Jupyter Notebook interface to amspython:

- First install and run Jupyterlab from the command-line. This will print a URL in the console. You can close the web browser window.
- In VS Code, choose Select Kernel → Existing Jupyter Server → enter the URL from the previous step.

Method 2: Create a conda environment that runs amspython

VS Code can typically autodetect conda environments on your machine. This lets you launch Jupyter notebooks directly from VS Code without first having to launch the server from the command-line.

These instructions assume that you have conda installed on your machine. If not, you can download and install it from for example Miniforge.

See also:

Get started with conda or mamba in the engine ASE documentation.

This has only been tested on Linux.

First, install Jupyterlab into amspython as above.

Then, create a small conda environment where the `python` executable simply runs `amspython`:

```
# create the conda environment
conda create -n amspython-env

# create the python executable
conda run -n amspython-env sh -c '
mkdir -p "$CONDA_PREFIX/bin"
PYTHONFILE="$CONDA_PREFIX/bin/python"
```

(continues on next page)

(continued from previous page)

```
[ -f "$PYTHONFILE" ] && echo "$PYTHONFILE already exists, I will not overwrite it." &&
↪ exit 1
echo "#!/bin/sh" > "$PYTHONFILE"
echo "\"\$AMSBIN/amspython\" \"\$@\"" >> "$PYTHONFILE"
chmod +x "$PYTHONFILE"
'

# test the python executable
conda run -n amspython-env python -c '
import sys
print(f"amspython-env runs python from {sys.executable}")
'
```

In VS Code, you can then Select Kernel → Select Other Kernel → amspython-env.

2.4.4 Execute a notebook from the command-line

If you want to run a Jupyter notebook from the command-line, for example when submitting to a cluster, you can do something like the following

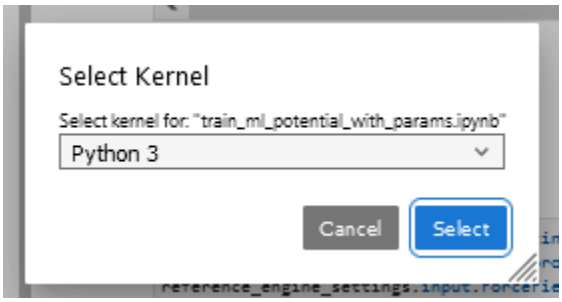
```
cat > template.tpl <<EOF
{%- block body %}
{{ nb | json_dumps }}
{% endblock body %}
EOF


"$AMSBIN/amspython" \
    -m nbconvert \
    --to custom \
    --template template.tpl \
    --RegexRemovePreprocessor.patterns='^!' \
    --execute \
    --output output.ipynb \
    notebook.ipynb
```

Above, the purpose of `template.tpl` is to allow the use of the `RegexRemovePreprocessor`. It will exclude cells starting with an exclamation mark `!`, for example commands used to launch GUI modules like AMSmovie or AMSspectra.

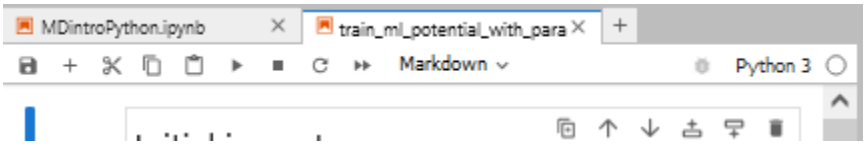
2.4.5 Jupyter frequently asked questions

Select kernel

When opening a notebook, you may be asked to select a kernel:



If unsure which is the correct kernel, create a new notebook in Jupyter and find the correct kernel in the top right:



Note that Jupyter may detect also kernels unrelated to AMS on the system, e.g. installed by another program or the user.

Notebook is not trusted

Notebooks not created by the user, including those downloaded from scm.com, will not always be trusted by Jupyter. A warning may display in the terminal saying `Notebook <path>/<filename>.ipynb is not trusted`. This means that Jupyter will not execute the contents automatically, but it is still possible to run the contents manually, e.g. through pressing shift-enter in a cell. To trust a notebook, either use the command `jupyter trust <path>/<filename>.ipynb` in the command line, or click `View > Active Command Palette` and type `Trust Notebook` in the search bar.

— Logging error —

On Windows, a `--- Logging error ---` is sometimes displayed. This happens when Unicode characters (e.g. powers or symbols related to units) are displayed. The error can be ignored, the characters will be displayed correctly in the notebook and the source in the affected cell is fully executed.

2.5 Python virtual environment

2.5.1 Default Python virtual environment

Starting with AMS2020, the `amspython` command by default checks for a Python virtual environment inside the user's home directory. If it does not find a virtual environment, it will create one in the following location (these are the default values for the `SCM_PYTHONDIR` environment variable):

- Windows: `$USERPROFILE/.scm/python`
- Mac: `$HOME/Library/Application Support/SCM/python`
- Linux: `$HOME/.scm/python`

`amspython` then launches the Python binary located inside the virtual environment.

If you install additional packages via pip (see above), they will be installed into the virtual environment's `site-packages`.

2.5.2 Virtual environments for different AMS versions

The virtual environment is tied to the major release version of AMS, which is reflected in the name of the virtual environment directory (e.g. AMS2020.1.venv).

Thus, upgrading from AMS2020.101 to AMS2020.102 will automatically let you use all Python packages that you installed into the AMS2020.101 virtual environment.

To use previously installed Python packages when upgrading from AMS2020.101 to e.g. AMS2020.301 or AMS2021.101, simply copy or rename AMS2020.1.venv to AMS2020.3.venv or AMS2021.1.venv in the same directory.

If you have several installations of different major releases of the Amsterdam Modeling Suite on the same computer, multiple virtual environments will also be created, one for each installation.

2.5.3 Changing the location of the Python virtual environment

Set the `SCM_PYTHONDIR` environment variable to a directory in which the virtual environment will be installed. If you use the graphical user interface (GUI), this environment variable can be changed in the GUI preferences. If you use the command line, set it in your `amsbashrc.sh`. If you use both the GUI and the command line, you should change it in both places.

2.5.4 Disabling the virtual environment

Follow the steps for changing the location of the Python virtual environment, but set `SCM_PYTHONDIR` to be empty.

2.5.5 Uninstalling the virtual environment

Just delete the directory containing the virtual environment. This will also delete any packages that you have installed into it.

2.5.6 Reinstalling the virtual environment

To force a reinstallation of the virtual environment, even if it already exists, run `$AMSBIN/amspython --install_venv`. This will not remove any packages inside the virtual environment.

2.6 Useful commands

- Start a Python shell:

```
$AMSBIN/amspython
```

- Install new packages via pip:

```
$AMSBIN/amspython -m pip install name_of_package
```

- Find out the location of an installed package (e.g. numpy):

```
$AMSBIN/amspython -c 'import numpy; print(numpy.__file__)'
```

- Find out which python binary is launched by amspython:

```
$AMSBIN/amspython -c 'import sys; print(sys.executable)'
```

- Disable the virtual environment (one-time):

```
SCM_PYTHONDIR='' $AMSBIN/amspython
```

CHAPTER

THREE

COMMAND LINE TOOLS

- **amsprep**: prepare an ADF job from a script (or command line).
- **amsreport**: get information (including images) from an ADF result file (for use in your script, or to generate an HTML or tab-separated report).
- **pkf**, **cpkf**, **dmpkf**, **udmpkf**: the KF utilities, which are command-line utilities to process KF files.

3.1 AMSprep: generate (multiple) ADF jobs

AMSprep allows one to generate input files for the different programs of the Amsterdam Modeling Suite by means of console commands. As such AMSprep can be used to run the same type of calculation on a series of different chemical systems. Another important example are automatic checks of the convergence of the results with respect to the computational parameters e.g. by varying input settings such as basis set choice or numerical integration accuracy while recomputing the same system.

AMSprepare ($AMSBIN/amsprep) generates a job script from a template .ams file. Such a template file can either be produced by AMSinput or simply be found among the default templates included. These default templates are identical to those present in AMSinput.

Two **examples** are presented here to demonstrate the capabilities of AMSprep:

- In BakersetSP you will see how to use amsprep to run a particular job for a test set of molecules. The individual molecular structures are provided as xyz-files which contain no ADF specific information. AMSreport is used to collect the values of the bonding energies resulting from these calculations.
- In ConvergenceTestCH4 you will see how to use AMSprep to test convergence of the bonding energy with respect to the basis set and the numerical integration grid.

The options of AMSprep are listed when running the module without further command line arguments, or with the -h flag:

```
% amsprep -h
AMSprepare (amsprep) generates a job script from a .ams file (the template),
with user specified changes to input options / method / system.

Usage: amsprep -t template.ams [-m molecule.(ams|adf|xyz|mol|t21)] [-z charge] [-s␣
↪spin]
                               [-runtype SinglePoint|GeometryOptimization|Frequencies]
                               [-gradientsonly]
                               [-q quality] [-zlmfit quality] [-kspace quality]
                               [-lattice v1.x v1.y v1.z ...]
                               [-i integration] [-b basis] [-c core] [-r relativity]
                               [-basiscacheid id]
```

(continues on next page)

(continued from previous page)

```
                                [-x xcpotential] [-e xcenergy] [-bondsonly]
                                [-dftbmodel DFTB|SCC-DFTB|DFTB3] [-dftbparameters dir]
                                [-dftbdispersion None|Default|D3-BJ|D2|ULG|UFF]
                                [-logfile logfile] [-j jobname] [-a amsfile]
                                [-dist "atom1 atom2 distance ..."]
                                [-angle "atom1 atom2 atom3 angle ..."]
                                [-dihed "atom1 atom2 atom3 atom4 angle ..."]
                                [-atomtype "atom type ..."]
                                [-structure "atom structure ..."]
                                [-pointcharges file]
                                [-efield "Ex Ey Ez"]
                                [-rxforcefield fname] [-rxniter n] [-rxnrstep n] [-
↪rxtstep T]
                                [-rxmethod method] [-rxmdtemp T] [-rxmdpres p]
                                [-region "name at1 at2... "]
                                [-fragments prefix] [-onejob]
                                [-g "key value"]


Start with a job template, adjust it for this particular job, and write the resulting␣
↪job
to standard output. Values specified should match exactly the values as you would␣
↪specify
using AMSinput, also for menu choices.

TEMPLATE
-t: the .ams file (saved by AMSinput) to be used as template, defining the whole job
    All other options override values from this job

    Instead of a .ams file, you may also specify the name of one of the standard␣
↪templates
    as defined in AMSinput: "Single Point", Frequencies, "Geometry Optimization", etc
    A special option for energy and gradients
    for the current geometry: EG (see also -gradientsonly)

    Some shortcuts: SP, EG, GO, FREQ, optionally prefixed by␣
↪(ADF|BAND|DFTB|UFF|MOPAC)-
    For example: ADF-FREQ, BAND-SP, DFTB-GO, MOPAC-EG

    Some ReaxFF shortcuts: REAXFF-EG for a single ReaxFF iteration

CHANGES TO TEMPLATE
-m: the molecule to use, element types and coordinates
    This can be taken from anything that AMSinput can import,
    for example .ams, .mol, xyz or .t21 files

    The -m flag may be repeated, each molecule added will be in its own region
    This may be used for fragment calculations, but it does not work with .ams files

    If you specify an .sdf file, you can select which frames to import:
        conformers.sdf#1-10         loop over the first 10 frames
        conformers.sdf#e2.0         loop over all frames with energy below 2.0
           (units as in the file, wrt the lowest energy of all frames in the file,
           energies from comment lines)
        conformers.sdf#1-10e2.0     loop over the first 10 frames,
                                    and use only those  with energy below 2.0
        conformers.sdf              use the first frame of the sdf file
```

(continues on next page)

(continued from previous page)

```
    If you specify a .t21 file, you can select which frames or range of frames to␣
↪import:
        ajob.t21#ircf3              3rd frame in the IRC forward path
        ajob.t21#ircb2              2nd frame in the IRC backward path
        ajob.t21#h7                 7th frame in the history
        ajob.t21#lt8                8th frame in the LT path
        ajob.t21#ircf3-10           IRCForward frame 3, 4, ... 10
        ajob.t21#ircf               IRCForward all frames, starting at 1
        ajob.t21#ircf0-             IRCForward all frames, starting at 0
                                    (original geometry, before first step)

    If you specify a .cry file, the compound to import may be specified:
        $AMSHOME/atomicdata/Molecules/Crystals/Cubic/CsCl.cry#MgTl

    When looping, all resulting jobs will be joined together, the jobname and ams␣
↪files
    get the frame sequence number appended after an _
    When looping only one -m flag may be specified

-xyz: use xyz coordinates from specified file, not touching anything else
      it is applied after -t and -m
      the elements and number of atoms should match
      currently works with KF and xyz files
-smiles: use smiles to describe the molecule

-irc: when using IRC frames in the -m flag, revert the backwards order

-dist: change the distance between atom1 and atom2 to the specified distance
       the arguments must be enclosed in quotes, and may be repeated for multiple␣
↪distances
-angle: change the angle (atom1, atom2, atom3) to the specified angle
       the arguments must be enclosed in quotes, and may be repeated for multiple␣
↪angles
-dihed: change the dihedral (atom1, atom2, atom3, atom4) to the specified angle
       the arguments must be enclosed in quotes, and may be repeated for multiple␣
↪angles
-atomtype: set the type (element) of atom to type
       the arguments must be enclosed in quotes, and may be repeated for multiple␣
↪types
-structure: add a structure just as if using the structure tool in AMSinput
       atom is the selected atom, structure is the name of the structure file
       the arguments must be enclosed in quotes, and may be repeated for multiple␣
↪changes
-liststructures: list available structure files for use with -structure, and exit

-runtype: run type (SinglePoint,GeometryOptimization,Frequencies)
-gradientsonly: after calculating the gradients, stop
                works also for excited state gradients if requested in your template
-z: charge (real number)
-s: spin (integer), if not zero this implies an unrestricted calculation
-q: quality (Basic, Normal, Good, VeryGood or Excellent), default for Becke/ZlmFit
-i: integration (integer)
-i: Becke integration (Basic, Normal, Good, VeryGood or Excellent)
-i: teVelde integration (integer)
-zlmfit: ZlmFit quality (Basic, Normal, Good, VeryGood or Excellent)
-kspace: KSpace quality (GammaOnly, Basic, Normal, Good, VeryGood or Excellent)
```

(continues on next page)

(continued from previous page)

```
-lattice: lattice vectors first three numbers for the first vector, next for the␣
↪second etc
          The dimension follows from the number of vectors
-b: basis type (SZ, DZ, DZP, TZ, TCP, TZ2P, QZ4P)
-c: core type (None, Small, Medium, Large)
-basiscacheid id: refer to t21 files from previous runs prefixed with this id
-r: relativistic level (None, Scalar, Spin-Orbit), using ZORA
-x: XC potential during SCF, one from the options available in AMSinput:
        LDA,
        GGA:BP, GGA:BLYP, GGA:PW91, GGA:mPW, GGA:PBE, GGA:RPBE, GGA:revPBE, GGA:mPBE,
        GGA:OLYP, GGA:OPBE,
        Model:SAOP, Model:LB94,
        Hartree-Fock,
        Hybrid:B3LYP, Hybrid:B3LYP*, Hybrid:B1LYP, Hybrid:KMLYP, Hybrid:O3LYP,␣
↪Hybrid:X3LYP,
        Hybrid:BHandH, Hybrid:BHandHLYP, Hybrid:B1PW91, Hybrid:MPW1PW, Hybrid:MPW1K,
        Hybrid:PBE0, Hybrid:OPBE0
-e: XC energy after SCF (Default, LDA+GGA_METAGGA, LDA+GGA+METAGGA+HYBRIDS)
-pointcharges: file, file with point charges, one point charge per line (ADF only)
               x y z charge, xyz in Angstrom, charge in elementary units (+1 for a␣
↪proton)
-efield: Ex Ey Ez the electric field vector (in Hartree/(e Bohr))
-k: replace any key, the single argument will be broken into:
     the key, the replacement value, and END for a block key
     all separated by spaces. To insert a return, add a |
     When the key is not found, it is added just before the ATOMS key
     The -k key may be repeated, and is applied at the end, replacing even earlier␣
↪changes

-dftbmodel DFTB|SCC-DFTB|DFTB3: select the DFTB model
-dftbparameters dir: select the directory with DFTB parameters
-dftbdispersion [None|Default|D3-BJ|D2|ULG|UFF]: dispersion option to use, default is␣
↪None

-rxforcefield fname: the ReaxFF force field file
-rxniter n: number of ReaxFF iterations
-rxnrstep n: number of non-reactive iterations (out of the total number of iterations)
-rxtstep T: the time step used in the MD simulation
-rxmethod string: the simulation type: Velocity Verlet + Berendsen|NPT|NVE
-rxmdtemp T: the thermostat temperature
-rxmdpres p: the required pressure

-region name at1 at2 ...: make a region with specified name and atoms, may be repeated
     The atom numbers at1 at2 refer to input order, after geometry modifications,␣
↪start at 1
     Use at1-at2 to refer to all atoms between at1 and including at2
     If the region key is present all regions already present are deleted
-fragments prefix: set up a fragment calculation, prefix fragment run/job scripts␣
↪with prefix
                   if this key is present fragment run/job scripts will be saved␣
↪(unless -onejob)
                   if a job script is requested, the fragment job names will be␣
↪prefix.fragname.job
 -onejob: for fragment jobs, concatenate the fragment jobs and final job into one on␣
↪stdout
-g "key value": set any key to the specified value (note key value within quotes)
         key: internal name in AMSinput for some option, see bin/amsinput.tcl/tpl/
```

(continues on next page)

(continued from previous page)

```
↪Defaults.tpl
         value: set gin(key) to the specified value
 -nochain: unset chain option (used internally by chain jobs)

OUTPUT
-bondsonly: only the bonds as generated by the GUI will be exported (the GUIBONDS␣
↪block)
-logfile: force the specified logfile to be used in the run script
-j: produce a fully runnable job (as the .job files from AMSjobs),
    using the specified jobname.
    The job script produces files like jobname.out, jobname.t21 etc. Several job␣
↪scripts can simply
    be concatenated, the results will be stored in different files using th jobname␣
↪parameter
    the default is a simple run script (the .run file from AMSinput, files are left␣
↪as they are)
-a: save a .ams file that matches the run script, except for the -k arguments
    (they are listed in the user input field)
    amsfile is the name of the AMSinput, including the .ams extension (required)

Example: calculate gradients for a molecule in file mymol.xyz
         amsprep -t GO -m mymol.xyz -k "stopafter ggrads"

Example: calculate gradients for a molecule in file mymol.xyz, using good quality␣
↪integration and fit:
         amsprep -t GO -q Good -m mymol.xyz -k "stopafter ggrads"

Example: calculate DFTB frequencies for a molecule in file mymol.xyz
         amsprep -t DFTB-FREQ -m mymol.xyz
```

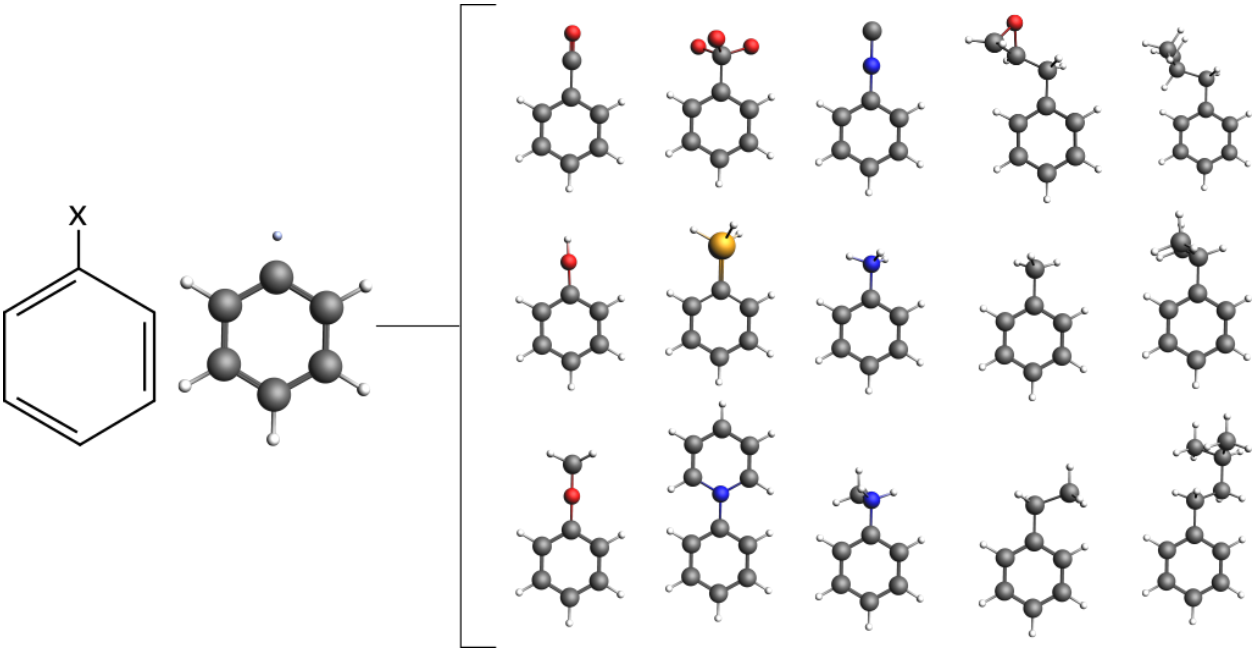
3.1.1 Additional Notes

CRSprep represents a scripting solution which is exclusively oriented towards generating input files for the COSMO-RS program.

3.2 Tutorial: Generate structures for substituent effects screening

Overview

Screening substituent patterns of a base compound is a common task in computer aided materials design. In the following short tutorial we demonstrate how you can use amsprep to automatize the replacement of substituents with just a few lines of simple shell scripting.

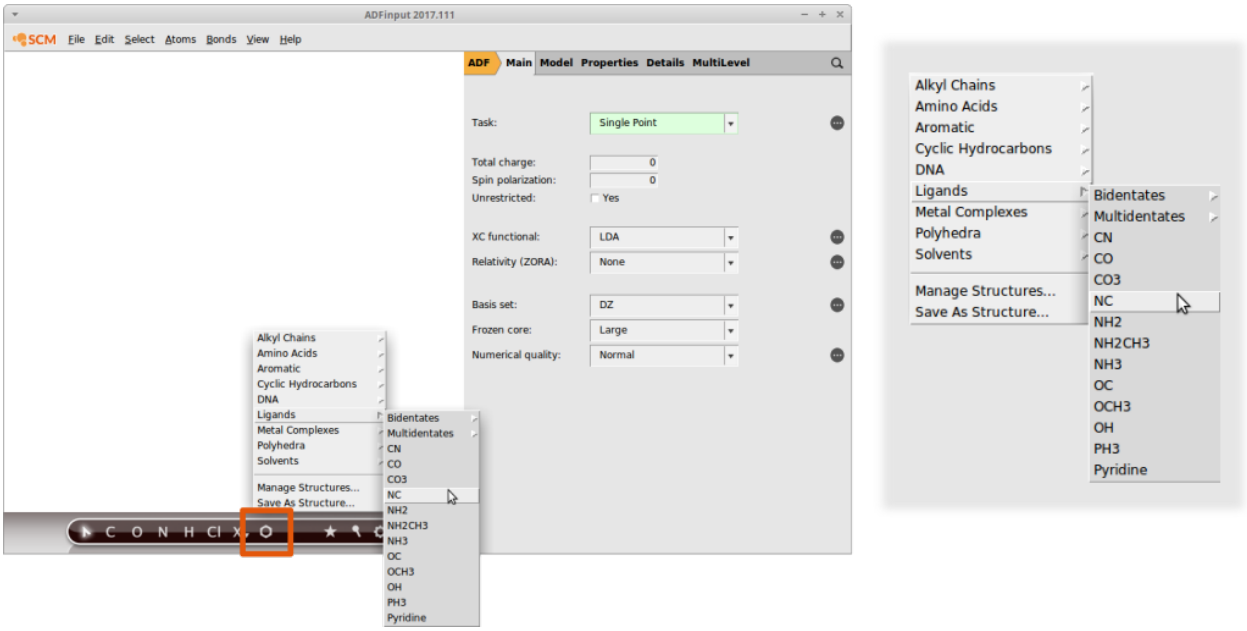


Contents:

- The library of substituents in AMSinput
- Exchanging substituents with AMSprep
- Combining AMSprep and AMSreport in shell script

The substituent library in AMSinput

AMSinput comes with a customizable library of common substituents that we can use for our screening purposes right away. It can be accessed via the structure builder tool in AMSinput:

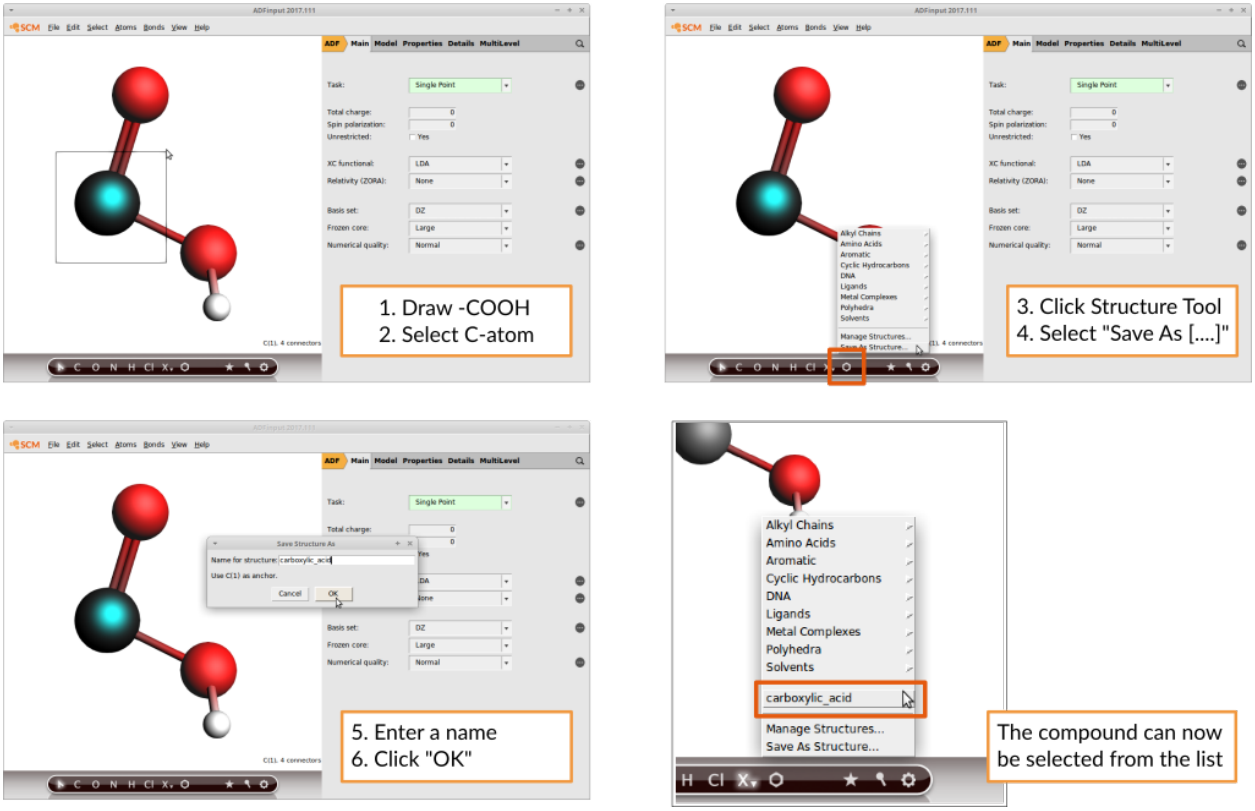


Note how the entries are organized. For example the isocyanide functional group (“NC”) can be found in “Ligands”.

Its also possible to add your own compounds: Simply draw the structure of interest and select the atom which will serve

as an anchor.

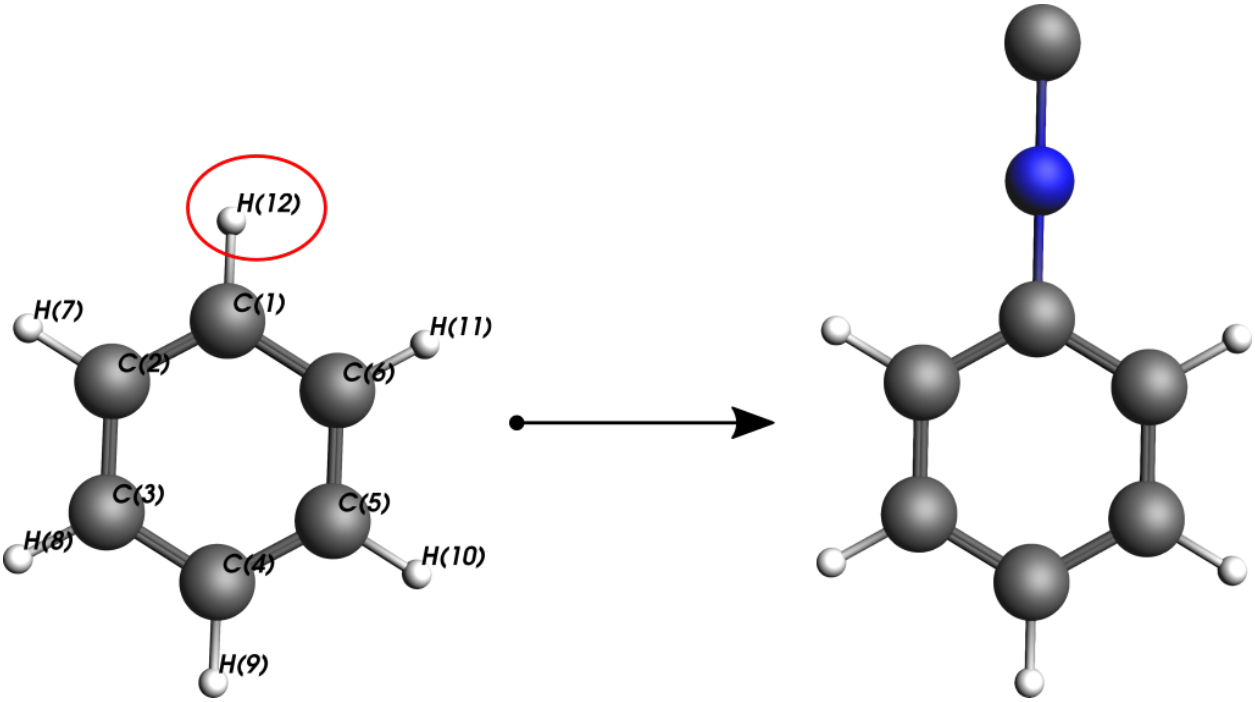
This is how you would add a carboxylic acid group:



AMSinput will always consider the atom that lies in the origin of the coordinate system to be the anchor. If you use the Save As Structure command this will be done for you. More information, including another example, can be found in the Building Molecules GUI tutorial.

Exchanging substituents with amsprep

Let us consider a simple benzene ring as the base compound:



The amsprep command to exchange Hydrogen atom #12 with an isocyanide group ("NC") and create a runfile for a UFF geometry optimization is:

```
"$AMSBIN/amsprep" -t UFF-GO -m benzene.xyz -structure "12 Ligands/NC.ams" > "benzene_
↪NC.run"
```

Remember that the "CN" group was located in the "Ligands" menu hence "Ligands/NC.ams". In case the path contains whitespace, you need to escape the whitespace as in this example

```
"$AMSBIN/amsprep" -t UFF-GO -m benzene.xyz -structure "12 Alkyl\ Chains/Ethyl.ams" >
↪"ethyl_benzene.run"
```

When using custom substituents, e.g. the hydroxylic_acid in the above example, a full path need to be provided to amsprep. The path is displayed when clicking on the Structure Tool in AMSinput and selecting "Manage your structures". On an ubuntu linux system the path is "/home/[your_username]/.scm_gui/Structures" and the command to use your own structures becomes:

```
"$AMSBIN/amsprep" -t UFF-GO -m benzene.xyz -structure "12 /home/[your_username]/.scm_
↪gui/Structures/carboxylic_acid.ams" > "benzoic_acid.run"
```

Bringing it all together

The following few lines of shell script demonstrate how to automatically exchange the substituents on a benzene ring, run a UFF optimization on the new structure and extract the optimized geometry with amsreport.

```
#! /bin/sh
#
# copy the file benzene.xyz from the ADF compounds database
#
cp "$AMSHOME/atomicdata/Molecules/ADF/Benzene.xyz" .
#
# loop through different substituents
#
```

(continues on next page)

(continued from previous page)

```
for ligand in CN CO CO3 NC NH2 NH2CH3 NH3 OC OCH3 OH PH3 Pyridine; do
   #
   # prepare the coordinates and the UFF calculation
   #
   "$AMSBIN/amsprep" -t UFF-GO -m Benzene.xyz -structure "12 Ligands/$ligand.ams" >
↪"benzene_$ligand.run"
   #
   # run UFF GeoOpt
   #
   sh "./benzene_$ligand.run"
   #
   # extract the optimized geometry via amsreport
   #
   "$AMSBIN/amsreport" uff.rkf SDF > "benzene_$ligand.mol"
   #
   # rename the generic UFF output file
   #
   mv uff.rkf "benzene_$ligand.rkf"
done
```

Running the script

Linux and Mac: Copy and paste the above into a file called `substituents_script` and execute it in the command line

```
sh substituents_script
```

Windows: Just use the pre-configured shell `ams_command_line.bat` shipped with ADF to run the same command as the Linux and Mac users.

3.3 AMSreport: generate reports

The utility AMSreport ($AMSBIN/amsreport) allows to retrieve the results (including images) computed from the binary output files of either ADF, BAND, ReaxFF, DFTB, UFF, or MOPAC. For ADF this is the .t21 file (TAPE21). It can also be the .runkf file from BAND, the .rxkf file from ReaxFF or the .rkf file from DFTB, MOPAC or UFF.

The selected results are printed out via standard output or, alternatively, either written to a tab separated file or an HTML file. When creating a new output file AMSreport will also generate a line with headers identifying the information. Images are generated using the ADF-GUI.

Also individual KF variables can be retrieved from the file as shown by the following example, which illustrates how to obtain the bonding energy from a .t21 file.

```
amsreport job.t21 BondingEnergy
```

Also high-quality pictures of orbitals can be obtained as shown below.

```
amsreport job.t21 HOMO LUMO+1 -v "-grid Fine" -v "-antialias" -v "-bgcolor #ffffff"
```

The options of AMSreport are listed when running the module without further command line arguments. At present the following command line options are available

-h
: prints the help screen.

Hint: If used with the name of a valid KF file in the command line the -h option lists the names of all data blocks present in that file. It is strongly encouraged to use this option to retrieve the names of the options available in a given situation.

```
amsreport -h job.t21
```

-i
specifies the input file (.t21 etc). If the specified input file is not present ADF tries to find a valid input file based on the information in the matching .ams file or the most recent available binary output file.

-usefile
specifies the input file like -i but without attempting to find a matching file if the specified input file does not exist. Typically -usefile is used to avoid reading data from the result file.

-I <pattern>
glob files, and run over all matching result files

-o
the name of the html file in which the output of AMSreport will be stored. The output will be printed to standard output if this option is absent.

-plain
print only output data from AMSreport without any labels and/or units. The same can be achieved by setting the environment variable SCM_AMSREPORT_PLAIN to yes.

-noplain
print output data with tab separators, labels, and units. Used to override the aforementioned variable SCM_AMSREPORT_PLAIN.

-v
command line to pass to amsview (without filenames) to generate images. The image will be generated by AMSview stored in a directory with a name based on the result file, and with extension .jpgs. The result file will contain a path to the image file (directly, or in an IMG tag) After the -v the arguments must be listed, with proper quoting. Repeat the -v flag for multiple arguments. The individual -scmgeometry, -bgcolor, -zoom, -viewplane, -antialias and -grid options will be collected and applied to all view options.

Some shortcuts are predefined (HOMO, HOMO+1, LUMO, Molecule, Density, Potential) and some additional useful flags include

-scmgeometry (default 200x200) -bgcolor (default #220000), -zoom (default 1.0) -viewplane (default {1 2 5}) -antialias (off when not present, especially useful with light bgcolors) -grid (Coarse when not present, Medium when specified, or value after flag if a value is present)

examples
HOMO-1 LUMO+1 -v "-viewplane {0 0 1}" -v "-grid Fine" -v "-antialias"

-r
Specifies the result to be retrieved by AMSreport from the binary output file. If this command is omitted all unspecified command line arguments but the first (denotes input file name) will be considered as arguments for this flag.

If -r is present, the desired result is specified as a string either in form of its preset name (see below) or via a section%variable pair (see the KF utilities documentation). The -r flag (or arguments without flag) may be repeated for multiple results. Additional details can be specified after the variable name, separated by "#". For example

range
"variable#index" or "variable#firstindex:lastindex", index starts at 1

format
TclTk format string, e.g. 8.3f or 12.6g

examples

prints a formatted table of the coordinates

```
-r "Geometry%xyz#12.4f##3"
```

prints a formatted table for the first two atoms only

```
-r "Geometry%xyz#1:9#12.4f##3"
```

coordinates of the first two atoms in one line

```
-r "Geometry%xyz#12.4f#1:9"
```

print just the first coordinate

```
-r "Geometry%xyz#1"
```

print the bond energy

```
-r "Energys%Bond Energy"
```

While any proper KF variable can be accessed via a "section%variable" construct, the following predefined keys are available for the KF files resulting from the various programs of the Amsterdam Modeling Suite.

ADF-specific ``-r`` presets for .t21 files

`orient*`
affine transform (3x4) from input to internal ADF orientation, format after #

`iorient*`
affine transform (3x4) from internal ADF to input orientation, format after #

`title`
title of the calculation

`type`
calculation type (single point, geometry optimization, …)

`weight`
molecular weight

`symmetry`
molecular symmetry

`natoms`
number of atoms

`integration`
integration accuracy

`integration-min`
minimum integration accuracy

`integration-max`
maximum integration accuracy

`scfstatus`
SCF convergence status

`charge`
the requested charge

charges
shorthand for Voronoi, Hirshfeld and Mulliken charges

voronoi
Voronoi deformation charges

hirshfeld
Hirshfeld fragment charges, atomic fragment definition required

mdc
All available MDC atom charges

mdc-m
MDC-M charges

mdc-d
MDC-D charges

mdc-q
MDC-Q charges

mulliken
Mulliken charges

bondorders
Mayer bond orders

nmr
NMR shieldings

nmr-shieldings
NMR shieldings

nmr-shielding-tensor
NMR shielding tensor

nmr-j-coupling-tensor
NMR j coupling tensor

nmr-k-coupling-tensor
NMR k coupling tensor

nmr-j-coupling-constant
NMR j coupling constant

nmr-k-coupling-constant
NMR k coupling constant

dipolev*
dipole vector

dipole
dipole moment (length of dipole vector)

quadrupole
quadrupole tensor

orbital-info
orbital info (energy, occupation and label), format for energy after #, range after # with HOMO or LUMO for example:

```
orbital-info#HOMO, orbital-info#HOMO-1,
orbital-info#HOMO-2:LUMO+2, orbital-info#HOMO#12.8f
```

orbital-e*
orbital energies, format and range after # as in orbital-info

orbital-o*
orbital occupations, format and range after # as in orbital-info

orbital-l*
orbital labels, format and range after # as in orbital-info

homo-lumo-gap*
HOMO-LUMO gap, format after #

atomlabels
name of atoms with sequence number, starting at 0

atomlabels-from0
name of atoms with sequence number, starting at 0

atomlabels-from1
name of atoms with sequence number, starting at 1

nstep
number of steps in history / LT / IRC data, type (h,lt,ircf,ircb) after #

spin
the requested spin polarization

step
use coordinates from history / LT / IRC data, step number after # with h for history, lt for LT, ircf/ircb for forward/backward IRC if no letter after #, history data will be used (if not, last step will be used) for example:

```
step#23 (or step#h23), step#lt4, step#ircf3
```

geometry, geometry-a*, geometry-b*
geometry (element type and coordinates), in input order, in angstrom or bohr (default)

sdf
geometry in SDF format

bgf
geometry in BGF format

distance
distance between two atoms, in angstrom. Input separated by #

labels (optional): include atom labels in output

format (optional): format field

atom numbers, starting at 1, in input order

examples

```
distance#2#3, distance#labels#2#3, distance#-8.3f#5#8,
distance#labels#8.4f#1#2, distance#2#3#4#5, distance#labels#1#2#3#4
```

angle
angle between three atoms, in degrees. Input see distance, but with three atoms per angle

dihedral
dihedral between four atoms, in degrees. Input see distance, but with our atoms per dihedral

hessian*
Hessian matrix (from GeoOpt%Hessian_CART), fmt and nperline options after #

gradients*
: gradients with respect to nuclear displacements (from GeoOpt%Gradients), fmt and nperline options after #

energies*
: all available energies (bonding up to xc, with labels), fmt option after #

bonding
: total bonding energy

pauli
: total pauli repulsion

steric
: total steric interaction

orbital
: total orbital interaction

electrostatic
: electrostatic energy

kinetic
: kinetic energy

coulomb
: electrostatic (steric and orbital interaction) energy

xc
: exchange-correlation energy

dispersion
: dispersion energy

frequencies*
: IR Frequencies, format, nperline and range (n, or n:n, start at 1) after #

freqint*
: IR Intensities, format, nperline and range (n, or n:n, start at 1) after #

freqlabel*
: IR Frequencies label (symmetry), format, nperline and range (n, or n:n, start at 1) after #

normalmode*
: normal modes (mass weighted), format, nperline and range (n, or n:n, start at 1) after #

zeropoint*
: zero-point energy

excitation*
: Excitation energies, format, nperline and range (n, or n:n, start at 1) after #

oscillatorstrength*
: Oscillator strengths for the excitation energies format, nperline and range (n, or n:n, start at 1) after #

excitlabel*
: Excitation labels (symmetry), format, nperline and range (n, or n:n, start at 1) after #

BAND specific ``-r`` presets for .runkf files

natoms
: number of atoms

geometry, geometry-a*, geometry-b*
: geometry (element type and coordinates), in input order, in angstrom or bohr (default)

sdf
geometry in SDF format

bgf
geometry in BGF format

distance
distance between two atoms, in angstrom. Input separated by #

labels (optional): include atom labels in output

format (optional): format field

atom numbers, starting at 1, in input order

examples

```
distance#2#3, distance#labels#2#3, distance#-8.3f#5#8,
distance#labels#8.4f#1#2, distance#2#3#4#5, distance#labels#1#2#3#4
```

angle
angle between three atoms, in degrees. #4 Input see distance, but with three atoms per angle

dihedral
dihedral between four atoms, in degrees. Input see distance, but with our atoms per dihedral

atomlabel, atomlabel-from0
name of atoms with sequence number, starting at 0

atomlabel-from1
name of atoms with sequence number, starting at 1

ReaxFF specific presets for .rxkf files

natoms
number of atoms

geometry, geometry-a*, geometry-b*
geometry (element type and coordinates), in input order, in angstrom or bohr (default)

distance
distance between two atoms, in angstrom. Input separated by #

labels (optional): include atom labels in output

format (optional): format field

atom numbers, starting at 1, in input order

examples

```
distance#2#3, distance#labels#2#3, distance#-8.3f#5#8,
distance#labels#8.4f#1#2, distance#2#3#4#5, distance#labels#1#2#3#4
```

angle
angle between three atoms, in degrees. #4 Input see distance, but with three atoms per angle

dihedral
dihedral between four atoms, in degrees. Input see distance, but with our atoms per dihedral

atomlabel, atomlabel-from0
name of atoms with sequence number, starting at 0

atomlabel-from1
name of atoms with sequence number, starting at 1

rx-frame n options
information for a particular reaxff frame. Note the spaces, you will need to quote this key.

```
n: frame number 0, 1, 2, ... (is not the ReaxFF step number)

options: combination of the following (if omitted, all will be reported)

   nframes: total number of frames

   step: the ReaxFF step number for the specified frame

   nats: number of atoms

   xyz: the xyz coordinates

   names: element names (C, H etc) for each atom in the same order as the␣
↪coordinates

   neighbors: bond information

   cell: cell information
```

example

```
amsreport water.rxkf "rx-frame 20 step xyz cell"
```

pdbtrajectory
the trajectory information (including molecule details) as a sequence of PDB models due to limitations of the PDB format to less than 100000 atoms and it will not be a standard conforming PDB file

```
pdbtrajectory-(nobonds|usepdbinfo)
```

```
nobonds: as pdbtrajectory, but no bond info (CONECT records)

usepdbinfo: as pdbtrajectory, but use pdb residue info from first step instead of␣
↪reaxff mol info

xmol: the trajectory information (only element, xyz) in xmol format

gro: trajectory as .gro file (xyz and velocities) options after a - sign:

   m : print list of molecule names and formulas only

   x : allow xyz only frames (missing velocities)

   f : add forces if available

   tf : add the time step, f is a floating point number that is the time per step in␣
↪ps

examples: gro-x, gro-f, gro-xf,  gro-ft0.0001, gro-xt0.001, etc.

Special features for ReaxFF parameter optimization: a geo file in biograph format can␣
↪be converted from a DFT result file using the bfg option above.
```

example
Input file: geo (biograph format)

-rxtrainset: run over frames in the input file (should be a bgf BIOGRAPH file), put all charges, bonds and angles

in the trainset.in (on stdout).

Input file: ffield (reaxff force field file). The source ffield file determines which atoms, bonds etc are present.

-ffield-min: generate ffield file with all values replaced by min values

-ffield-max: generate ffield file with all values replaced by max values

-ffield-bool: generate ffield file with all values replaced by bool values

-minmax filename: use data from filename for min and max values,

format: see RxParRange.txt in atomicdata/ForceFields/ReaxFF

General presets for .rkf files

`natoms`
number of atoms

`geometry, geometry-a*, geometry-b*`
geometry (element type and coordinates), in input order, in angstrom or bohr (default)

`sdf`
geometry in SDF format

`bgf`
geometry in BGF format

`distance`
distance between two atoms, in angstrom. Input separated by #

labels (optional): include atom labels in output

format (optional): format field

atom numbers, starting at 1, in input order

examples

```
distance#2#3, distance#labels#2#3, distance#-8.3f#5#8,
distance#labels#8.4f#1#2, distance#2#3#4#5, distance#labels#1#2#3#4
```

`angle`
angle between three atoms, in degrees. #4 Input see distance, but with three atoms per angle

`dihedral`
dihedral between four atoms, in degrees. Input see distance, but with our atoms per dihedral

`hessian*`
Hessian matrix (from GeoOpt%Hessian_CART), fmt and nperline options after #

`gradients*`
gradients with respect to nuclear displacements (from GeoOpt%Gradients), fmt and nperline options after #

`energies`
all available energies (bonding up to xc, with labels), fmt option after #

3.3.1 Additional notes

- SDF and BGF records can be produced from from ANY file that can be read by AMSinput.
- KFreader is a free (LGPL) alternative to AMSreport. The C sources are available in our download section (http://www.scm.com/Downloads/KFReader-20140106.zip) and can be modified for more specific needs.

3.4 KF command line utilities

There are four utility programs for manipulating files in the so-called Keyed File (KF) format from the command shell. Two of them convert KF files from binary to ASCII and vice versa. See the pkf and dmpkf utilities for a description of the ASCII format of a kf file. Such a readable version of a KF file can be useful to inspect its contents in detail.

All programs from the package will convert a KF file to the binary format native to this platform if necessary. In such a case, the original file will be renamed to a file with tilde "~" appended to its name and a message will printed on the standard output.

The KF software was developed at the Vrije Universiteit Amsterdam as a general-purpose package for storing data and re-accessing it via keyword-driven procedures.

pkf

```
pkf file1 { file2 ... filen }
```

pkf prints a summary of the contents of the kf files file1… filen.

All variables are listed by name, type (integer, real, character, logical), and size (number of array elements) and bundled into named sections.

To put the results in an ASCII file for later inspection:

```
pkf file > ascii_result
```

Each section on the file contains an index of its variables and their associated values. All data are organized in blocks. Each section may have any number of index blocks and any number of data blocks (this depends simply on the amount of data to be stored in such a block). In addition there is one special section, the SuperIndex, which is an index of all sections on the file.

The output of pkf consists of:

- General information about the file (name of the file, internally used unit numbers during processing the file…)
- A summary of the SuperIndex, hence an index of blocks in the file and the associated sections.
- A summary: total numbers of blocks associated with the different types of blocks.
- For each section a list of its variables. For each variable in the list the following is displayed
  - The variable name.
  - Its length, i.e. the storage requirements of the variable within the file.
  - Its 'used' size, hence the file storage associated with the variable (in units off 8 Bytes for double precision real numbers, 4 for integers, etc.).
  - The number of actual elements within the variable (for real, integer, and logical data types) or the number of characters in a string.
  - The (logical) index of the data block it is stored in.
  - The off-set of the data within its data block.

  - Its value or the first element of an array variable, respectively.

cpkf

```
cpkf file1  file2 {key1 .. keyn}
```

cpkf copies the sections and/or variables key1 .. keyn from file1 to file2.

If a referenced section or variable already exists on file2 it is overwritten, else it is created. Sections and variables which are already present on file2 but which are not referenced in the command are not affected.

If no sections and/or variables are explicitly mentioned at all the copying is carried out for all sections and variables on file1.

As a side effect of this operation any 'holes' eventually present in the original due to the formal deletion of obsolete sections and variables are not copied. Note that the KF file is not rearranged upon deletion of data. Rather only the corresponding entries in the index tables are removed in this case. During the copying process the data is however rearranged for optimum storage efficiency and the resulting file copy may therefore be smaller than the corresponding original.

Skipping specific sections during the copying process can be manually controlled as follows:

```
cpkf file1 file2 -rm section1 ...
```

In this form, all sections will be copied except for the ones specified on the command line, thus effectively removing them from the file.

To copy and rename a section:

```
cpkf file1 file2  "section_name --rename new_section_name"
```

dmpkf

A utility to extract information from a KF file and make it available in ASCII format:

```
dmpkf file {key1 .. keyn}
```

dmpkf prints the sections and/or variables from the file file indicated by key1 .. keyn on standard output. The complete file is printed if no sections or variables are specified.

The format to be used for the individual keys:

```
Sec%Var
```

where Var the variable of interest present in section Sec. The complete section is dumped if no variable name is specified.

By redirecting the result to another file a human readable output is obtained:

```
dmpkf file > ascii_result
```

The output contains for each printed variable:

- One line with the name of the section it belongs to;
- One line with the name of the variable itself;
- One line with three integers:
  - The amount of space reserved for the variable on the file which is, however, relevant for programs operating with KF files only;
  - The amount of data associated with the variable: for reals, integers, logicals: the number of such elements; for strings: the number of characters;

  - An integer code for the data type of the variable: 1=integer, 2=real, 3=character, 4=logical;
- The values of the variable (on as many lines as necessary): for scalar variables only one value, for arrays as many values as the array contains.

udmpkf

A utility to put information read from standard input into a KF file:

```
udmpkf file
```

udmpkf reads an ASCII file in the format created by dmpkf from standard input and creates the binary KF file therefrom. If such a KF file is already present the sections and variables in the input file are appended to the existing KF file. Whenever a section or variable already exists in target file it will be overwritten. Other data on the target file are not affected.

The combination of dmpkf and udmpkf makes it easy to modify KF files with a normal text editor:

```
dmpkf TAPE21 > t21_ASCII
```

After the desired modifications within t21_ASCII this file may be reconverted into a binary KF file:

```
udmpkf < t21_ASCII TAPE21_new
```

Also note that dmpkf and udmpkf only require a single argument here, respectively, as "< t21_ASCII" passes the content of the edited file via the standard input.

akf

The `akf` utility is the command line version of the `AKFReader` python library described here: *AKFReader* (page 101).

For more information, refer to the help function of the `akf` tool:

```
akf --help
```

which we display below for convenience:

```
usage: -c [-h] [-c] [-cc] [-v] [-a] [-pa] [-ps] [-j] [-pv PRINT_VARIABLE [PRINT_
↪VARIABLE ...]] [-n] file [file ...]

A utility tool for Annotated KF files (akf).

positional arguments:
  file                  Path to a kf file of a folder (if a folder is specified, the␣
↪tool will recursively loop over all files with the '.rkf' extension)

optional arguments:
  -h, --help            show this help message and exit
  -c, --check           Check (i.e. validate) a kf file
  -cc, --check-conditional
                        Check (i.e. validate) a kf file, but only if the calculation␣
↪ended with 'normal termination' status
  -v, --verbose         verbose printing to stdout
  -a, --augment         include the kf definitions to the file
  -pa, --print-all      Print all the values in the kf file
  -ps, --print-skeleton
                        print skeleton
  -j, --check-json      Check a JSON definition file for validity instead of checking␣
↪a KF file against a JSON definition. This can be used to check for obvious errors␣
↪when writing the JSON definition
                        files for KF contents.
```

(continues on next page)

(continued from previous page)

```
  -pv PRINT_VARIABLE [PRINT_VARIABLE ...], --print-variable PRINT_VARIABLE [PRINT_
↪VARIABLE ...]
                        Print a variable. Note: you cannot have additional optional␣
↪arguments after this. If you want to have multiple optional arguments, specify them␣
↪before --print-variable
  -n, --no-metadata     Do not print metadata information, such as comments, shapes,␣
↪etc... (only relevant for print commands)

Example usage:
  akf path_to_file.rkf --check --augment
  akf path_to_file.rkf --print-skeleton
  akf path_to_file.rkf --no-metadata --print-variable Molecule%Coords
```

CHAPTER

FOUR

SCM LIBBASE

The module `scm.libbase`

4.1 Chemical System

4.1.1 Overview

The ChemicalSystem class serves as a versatile representation of a chemical system. It's designed to handle various types of chemical structures, such as molecules, surfaces, or crystals.

See the *UnifiedChemicalSystem* (page 83) API docs for a complete overview.

Here's how you can initialize a ChemicalSystem object using a System Block string:

```
from scm.libbase import UnifiedChemicalSystem as ChemicalSystem

# Initialize a Chemical System from a 'System Block' string
mol = ChemicalSystem(
"""
System
     Atoms
          O 0.0 0.0 0.0
          H 1.0 0.0 0.0
          H 0.0 1.0 0.0
     End
End
""")

# Guess the bonds in the molecule based on the atomic coordinates
mol.guess_bonds()

# Translate the molecule so that the origin coincides with its center of mass
mol.translate(-mol.center_of_mass())

# Print the molecule in 'System Block' format
print(mol)
```

4.1.2 Reading and writing, File formats

File formats overview:

Format	Read	Write	Note
.in (AMS System block)	*__init__()* (page 83)	*__str__()* (page 84)	Lossless serialization, includes bonds
.in (AMS System block)	*from_in()* (page 87)	*write_in()* (page 97)	Lossless serialization, includes bonds
.kf, .rkf (AMS binary file)	*from_kf()* (page 87)	*write_kf()* (page 97)	Lossless serialization, includes bonds
.xyz (plain)	*from_xyz()* (page 87)	`to_xyz()`	May give rounding errors, does not include bonds, lattice, charge or atomic properties
.xyz (AMS extended)	*from_xyz()* (page 87)	`to_xyz()`	See here. May give rounding errors, does not include bonds
.xyz (ASE extended)	No	No	
.cif	No	No	Convert to/from PLAMS Molecule for .cif files
.mol2	No	No	Convert to/from PLAMS Molecule for .mol2 files

You can create or serialize a ChemicalSystem object using various file formats. Among these, **System Block** is one of the most significant, offering a versatile text-based way to describe your chemical system. For more information on the syntax and options for the System Block, see the AMS System Block documentation.

All text based formats can also be written using Python format strings:

class UnifiedChemicalSystem

class UnifiedChemicalSystem(*system_block: str*)

A class representing a chemical system in the Amsterdam Modeling Suite.

__format__(*format_spec: str*) → str

Formats a ChemicalSystem into a string representation.

The `format_spec` string starts with and identifier for the format to write, followed and optional *:* and a list of space separated `key=value` pairs configuring options of the format. E.g. the following would produce the AMS System block format in internal units (bohr) with the string "H2O" as a System name in the block header:

```
cs = ChemicalSystem(...)
print(f"{cs:in:units=internal name=H2O}")
```

Would produce the output:

```
System H2O
    Atoms [bohr]
        O 0 0 -0.7262847342654172
        H 0 1.4669943905469853 0.3631423765813393
        H 0 -1.4669943905469853 0.3631423765813393
    End
End
```

The following formats and options are supported at the moment:

- `in` for writing the AMS System block. This is the default format, if none is specified. The following options are supported:

  - `name=...` to put an arbitrary string as the system's name into the block header.
  - `units=[default|internal]` to switch between default units and the units used by the ChemicalSystem internally. Internally the ChemicalSystem uses atomic units (e.g. bohr for lengths). Printing the System block in internal units avoids a possible loss in precision in the unit conversion and guarantees an exact `ChemicalSystem -> str -> ChemicalSystem` round-trip, i.e.:

```
cs = ChemicalSystem(...)
assert ChemicalSystem(f"{cs:in:units=internal}") == cs
```

  - `skip=[gui|adf|band|forcefield|dftb|reaxff|qe|...]` to avoid printing of a particular property group in the end-of-line string of an atom in the System%Atoms subblock. Multiple groups may be specified as a comma separated list.
  - `unused_atomic_properties=[drop|include]` to determine whether unused atomic property groups should be written out as `Modify%EnableAtomicProperties` entries. The default is not to do this, aka `drop`. This means that unused atomic property groups get lost in the `ChemicalSystem -> str -> ChemicalSystem` round-trip. If preserving them is imporant, this can be achieved by setting this option to `include`.

- `xyz` for writing a plain XYZ file without lattice or atomic properties. This format has no options.
- `extended_xyz` for writing the AMS extended XYZ format. This format has no options.

When you convert a ChemicalSystem object into a string (either by explicitly calling `str(my_chemical_system)` or by using a print statement) the output will be in the System Block format.

```
>>> from scm.libbase import UnifiedChemicalSystem as ChemicalSystem

>>> # Read it from a text file in the 'System Block' format:
>>> my_chemical_system = ChemicalSystem.from_in(filename="water.in")

>>> print(my_chemical_system)

System
    Atoms
        O     0.0000000000000000  0.0000000000000000  0.0000000000000000
        H     1.0000000000000000  0.0000000000000000  0.0000000000000000
        H     0.0000000000000000  1.0000000000000000  0.0000000000000000
    End
End

>>> # Write it to a file in the 'System Block' format:
>>> my_chemical_system.write_in(filename="another_water.in")
```

Note on lossless serialization

When you read or write a ChemicalSystem using either the System Block format or a *kf file* (page 80), the object is perfectly serialized. In other words, writing the object to a kf file and reading it back will result in an identical ChemicalSystem.

However, be cautious when using the xyz format as it doesn't offer lossless serialization. Writing and reading back using this format may result in the loss of certain information, such as bonds between atoms.

4.1.3 Elements

There are two classes for handling the elements of atoms in ChemicalSystems: *UnifiedElements* (page 99) and *UnifiedElement* (page 98).

- *UnifiedElements* (page 99) contains read-only static attributes for each element supported in AMS, accessible by their capitalized names. For example:
- `UnifiedElements.Hydrogen`: Represents the element Hydrogen.
- `UnifiedElements.Carbon`: Represents the element Carbon.

Each attribute is an instance of *UnifiedElement* (page 98) corresponding to that element. Additionally, the *UnifiedElements* (page 99) class also provides class methods to retrieve the elements by symbol or atomic number.

4.1.4 Atoms

The ChemicalSystem class maintains an array of instances of the `UnifiedAtom` class to represent the atoms in the system.

Adding and Modifying Atoms

The following code provides examples for adding atoms to a ChemicalSystem and modifying their properties.

```
from scm.libbase import UnifiedChemicalSystem as ChemicalSystem
from scm.libbase import UnifiedAtom as Atom

mol = ChemicalSystem()

# Three different ways of adding atoms
mol.add_atom("O", coords=[0, 0, 0])
mol.add_atom(Atom("H"), coords=[2, 0, 0])
mol.add_atom(1, coords=[0, 2, 0], unit="Angstrom")

# Change the coordinates of the Oxygen atom (units: Bohr)
mol.atoms[0].coords = [1, 2, 3]

# This is equivalent to the line above:
mol.coords[0, :] = [1, 2, 3]

# Change the first hydrogen atom to deuterium by adjusting the mass (units: Dalton):
mol.atoms[1].mass = 2.014
```

Available Methods

Here are methods to interact with the atoms in your ChemicalSystem:

class UnifiedChemicalSystem

class UnifiedChemicalSystem(*system_block: str*)

A class representing a chemical system in the Amsterdam Modeling Suite.

property atoms: UnifiedAtomList

A list of all Atom instances that are part of this ChemicalSystem.

property coords: AngstromCoordsArray

The coordinates of the atoms in angstrom.

contains_atom(*atom: UnifiedAtom*) → bool

Checks if an Atom instance is part of a ChemicalSystem.

atom_index (*atom: UnifiedAtom*) → int

Given an Atom instance, returns its index in ChemicalSystem.atoms.

add_atom (*atom: UnifiedAtom*) → None

add_atom (*atom: UnifiedAtom*, *coords: ArrayLike*, *unit: str = 'angstrom'*) → None

add_atom (*Z: int*, *coords: ArrayLike*, *unit: str = 'angstrom'*) → None

add_atom (*element:* UnifiedElement *(page 98)*, *coords: ArrayLike*, *unit: str = 'angstrom'*) → None

add_atom (*symbol: UnifiedAtom.T_AtomSymbol*, *coords: ArrayLike*, *unit: str = 'angstrom'*) → None

Add a new Atom to the ChemicalSystem.

remove_atom (*atom_index: int*) → None

Removes a single atom from the system, given its index.

remove_atoms (*atom_indices: ArrayLike*) → None

Removes multiple atoms from the system, given their atom indices.

set_atom (*atom_index: int*, *atom: UnifiedAtom*) → None

Safe assignment of an Atom instance to atoms[atom_index].

Is essentially the same as `self.atoms[atom_index] = atom`, but makes sure that all the atomic properties carried by the Atom instance are enabled on the target ChemicalSystem: the direct assignment will throw a ChemicalSystemError if the needed atomic properties are not enabled, while calling this method will enable them automatically and will not throw.

If Atom is part of a ChemicalSystem, its will overwrite whatever coordinates `atoms[atom_index]` had before. If Atom is not part of a ChemicalSystem, the coordinates of `atoms[atom_index]` will remain unchanged.

Methods for changing the order of atoms within a ChemicalSystem:

class UnifiedChemicalSystem

class UnifiedChemicalSystem (*system_block: str*)

A class representing a chemical system in the Amsterdam Modeling Suite.

reorder_atoms (*atom_order: List[int]*) → None

Reorders the system's atoms given a list of atom indices in the desired order.

The `atom_order` shall be a list of integers. The list shall be of length `num_atoms` and its values a permutation of `range(0,num_atoms)`. It shall contain the indices of the atoms in the original system in the order in which the atoms should occur after reordering.

Example: for a 6 atom system the `atom_order = [4, 5, 0, 1, 2, 3]` would have the effect of moving the last two atoms to the front of the system. (One might also look at `atom_order` as a mapping from the atom index in the reordered system to the atom index in the original system. Using the above example `atom_order[1] == 5`, which is the index that the new atom at index `1` had in the original system.)

sorted_atom_order (*comp: atom_comparator_func_t*) → List[int]

Returns the atom order based on a user defined comparator function as a list of atom indices.

The returned list can be passed into the `reorder_atoms` method to actually sort the atoms according to `comp`. See also documentation of the `reorder_atoms` and `sort_atoms` method.

sort_atoms (*comp: atom_comparator_func_t*) → None

Sorts the system's atoms according to a user defined comparator function.

Atoms are sorted in ascending order where the user defined comparator function implements the < operator.

Example for sorting atoms descending in their atomic number, i.e. heavy atoms first:

```
chemsys.sort_atoms(lambda A, B: A.Z > B.Z)
```

Note that the atom sorting is stable: two atoms for which `comp(A,B)` returns False will keep their relative order in the sorted system.

Methods for combining two ChemicalSystems into one:

class UnifiedChemicalSystem

class UnifiedChemicalSystem (*system_block: str*)

A class representing a chemical system in the Amsterdam Modeling Suite.

add_other (*other:* UnifiedChemicalSystem *(page 83)*) → None

Merges another ChemicalSystem into this one.

The number of atoms of `self` increases by the number of atoms of `other`. All atomic coordinates, properties and bonds between atoms will be kept. The total charge of `other` will be added to the total charge of `self`.

The regions of each atom do not change in the process. Regions with the same name in `self` and `other` are merged.

Systems with a lattice can only be merged with systems having a compatible lattice or no lattice at all. Lattices compatibility is checked with the `lattice.is_close()` method. If both systems have a lattice, and the lattice is compatible but not exactly the same, the lattice of the original system (`self`) is kept. If only one side has a lattice, that side determines the lattice of the result.

If both sides have a compatible lattice and bonds, merging them is only supported if either both or none of the two have the lattice displacements of the bonds set. If one side has lattice displacements, and the other does not, a ChemicalSystemError is raised.

__iadd__ (*arg0:* UnifiedChemicalSystem *(page 83)*) → *UnifiedChemicalSystem* (page 83)

Merges another ChemicalSystem into this one.

Note that lhs += rhs is just the operator version of lhs.add_other(rhs). See ChemicalSystem.add_other for details about merging systems.

__add__ (*arg0:* UnifiedChemicalSystem *(page 83)*) → *UnifiedChemicalSystem* (page 83)

Creates a new ChemicalSystem by merging two others.

Note that C = A + B is equivalent to C = copy(A); C.add_other(B). See ChemicalSystem.add_other for details about merging systems.

Splitting of ChemicalSystems into parts:

class UnifiedChemicalSystem

class UnifiedChemicalSystem (*system_block: str*)

A class representing a chemical system in the Amsterdam Modeling Suite.

extract_atoms (*atom_indices: ArrayLike*) → *UnifiedChemicalSystem* (page 83)

Returns a new system build from a subset of atoms.

Bonds within the subset will be preserved, but bonds to atoms not extracted will disappear. The returned system will have the same lattice as the original and a total charge of zero. Atomic properties and regions of the extracted atoms are preserved, but the returned system does not have any selected atoms.

split (*part_indices: ArrayLike*) → List[*UnifiedChemicalSystem* (page 83)]

Splits the system into parts and returns a list of these parts as separate systems.

Accepts a `num_atoms` long sequence, assigning the atoms of the system to the parts. The length of the returned list of parts is `max(part_indices)+1`.

Example: for a 6 atom system and a `part_indices` of `[0, 0, 0, 1, 1, 1]` a list of two systems will be returned. The system will contain the first three atoms of the original, and the second system the other three atoms.

The returned systems will have the same lattice as the original and a total charge of zero. Atomic properties and regions are preserved, but the returned systems do not have any selected atoms.

The UnifiedAtom class

The UnifiedAtom class contains all the relevant data for an individual atom:

class UnifiedAtom

class UnifiedAtom(*Z: int*)

class UnifiedAtom(*element:* UnifiedElement *(page 98)*)

class UnifiedAtom(*symbol: T_AtomSymbol*)

Class representing a single atom and its properties.

copy() → UnifiedAtom

Creates a deep copy of the Atom.

has_identical_properties(*other: UnifiedAtom*) → bool

has_identical_properties(*other: UnifiedAtom*, *properties: List[T_UnifiedPropertyGroup]*) → bool

Helper for @overload to raise when called.

in_chemicalsystem() → bool

Check if this Atom instance is part of a ChemicalSystem.

transmute(*Z: int*)

transmute(*element:* UnifiedElement *(page 98)*)

transmute(*symbol: T_AtomSymbol*)

Helper for @overload to raise when called.

property Z: int

The atomic number of the atom.

property adf: *UnifiedADFProperties* (page 43) | None

Atomic properties used by the ADF engine.

property band: *UnifiedBANDProperties* (page 45) | None

Atomic properties used by the AMS BAND engine.

property coords: AngstromCoordsArray

The coordinates of the atom in angstrom.

property dftb: *UnifiedDFTBProperties* (page 45) | None

Atomic properties used by the AMS DFTB engine.

property element: *UnifiedElement* (page 98)

Returns the atom's element.

property forcefield: *UnifiedForcefieldProperties* (page 45) | None

Atomic properties used by the AMS Forcefield engine.

property gui: *UnifiedGUIProperties* (page 46) | None

Atomic properties used for visualization in the GUI.

property is_ghost: bool

Whether the atom is a Ghost atom.

Ghost atoms are a special construct in AMS for calculating basis set superposition errors. Please refer to the ADF and BAND manuals for details.

property mass: float

The mass of the atom in dalton.

property qe: UnifiedQEProperties | None

Atomic properties used by the AMS QuantumESPRESSO engine.

property reaxff: *UnifiedReaxFFProperties* (page 45) | None

Atomic properties used by the AMS ReaxFF engine.

property symbol: T_AtomSymbol

The atom's symbol (e.g. 'C' or 'Au'). May be 'Gh.C' for ghost atoms.

Atomic species

One often needs to group a system's atoms into sets of indistinguishable atoms, essentially answering the question: which atoms are considered the same?

In AMS these groups of equivalent atoms are often called "atomic species". Note that the assignment of atoms into species is application dependent: an electronic structure code may only care about the elements of the atoms, while a molecular dynamics driver will also need to distinguish the different isotopes of the same element. Engines implementing classical forcefields will even need to distinguish atoms based on their forcefield atom types and formal charges.

The ChemicalSystem class allows users to provide a binary predicate function that implements the comparison of two atoms, to allow for any definition of atomic species:

class UnifiedChemicalSystem

class UnifiedChemicalSystem(*system_block: str*)

A class representing a chemical system in the Amsterdam Modeling Suite.

determine_species(*comp: atom_comparator_func_t*) → Tuple[int, List[int], List[int]]

Determines the present atomic species, based on a user defined comparator function.

The comparator shall return True for two atoms if they are to be considered of the same species.

Returns a 3-tuple consisting of:

1. `num_species` is the total number of detected distinct species.
2. `species_prototypes` a list of size `num_species` with the index of the first atom in the system that was recognized as being of a distinct species. This atom is considered the species' prototype atom and all atoms of the same species compare equal to it by the user defined comparator function `comp`.
3. `atom_to_species` a list of size `num_atoms` mapping each atom to its species.

See also the related `get_species` function, which returns the same data as a list of atom indices for each species.

In practice is usually easiest to implement the comparison as a lambda function right at the call site:

```
# Distinguish isotopes as separate species:
... = chemsys.determine_species(lambda A, B: A.Z == B.Z and A.mass == B.mass)
```

Note that you can access the ChemicalSystem instance from within the lambda, allowing access to things like *regions* (page 62) or *bonds* (page 51), that are not part of the *Atom* (page 39) class itself.

```
# Elements as species, but separately for region "substrate" and the rest:
... = chemsys.determine_species(
     lambda A, B: A.Z == B.Z and
     chemsys.is_atom_in_region(A, "substrate") == chemsys.is_atom_in_region(B,
↪"substrate")
)

# Distinguish species based on the number of bonds
# e.g. all 4-bonded atoms are equivalent:
... = chemsys.determine_species(
   lambda A, B: chemsys.bonds.num_bonds(chemsys.atom_index(A)) == chemsys.bonds.num_
↪bonds(chemsys.atom_index(B))
)
```

If you primarily want to iterate over all atoms within a species, the `get_species` function may return the same information in a more convenient format:

class UnifiedChemicalSystem

class UnifiedChemicalSystem(*system_block: str*)

A class representing a chemical system in the Amsterdam Modeling Suite.

get_species(*comp: atom_comparator_func_t*) → List[List[int]]

Determines the present atomic species, based on a user defined comparator function.

The comparator shall return True for two atoms if they are to be considered of the same species.

Returns a list of atom indices for each species, allowing for easy iteration over all atoms belonging to a species. Atom indices within each species are in ascending order.

```
for species in chemsys.get_species():
    for iat in species:
        # do something with chemsys.atoms[iat]
```

See also the related `determine_species` function, which returns the same information in a different format.

4.1.5 Atomic Properties

Atomic properties provide additional, customizable information for each atom. They are organized into categories, each relevant to different computation engines or modules.

- Generic Properties: These are common to all atoms, like mass.
- Engine/Module-Specific Properties: Specific to certain computation engines or modules. For example, properties relevant only to the ADF engine would be in the adfprops group and prefixed with 'adf' in the System Block.

You can toggle these property groups on or off. Here's an example, where we initialize some atomic properties in the System Block:

```
from scm.libbase import UnifiedChemicalSystem as ChemicalSystem

mol = ChemicalSystem(
"""
System
     Atoms
          O 0.0 0.0 0.0 forcefield.type=O_water
          H 1.0 0.0 0.0 forcefield.type=H_water mass=2.014
```

(continues on next page)

(continued from previous page)

```
          H 0.0 1.0 0.0 forcefield.type=H_water
     End
End
"""
)

# The Generic Properties are always available (and have default values if not␣
↪specified)
print(mol.atoms[1].mass) # Outputs 2.014
print(mol.atoms[2].mass) # Outputs 1.00798

# The forcefield properties are enabled because there were defined in the system block
print(mol.atomic_properties_enabled('forcefield')) # Outputs True

# Accessing the forcefield atomic properties
print(mol.atoms[0].forcefield.type) # Outputs O_water

# The adf properties are not enabled
print(mol.atomic_properties_enabled('adf')) # Outputs False

# Enable the 'adf' property group before using it:
mol.enable_atomic_properties("adf")
mol.atoms[0].adf.f = "fragment_1"
```

List of properties and utility methods in the ChemicalSystem:

class UnifiedChemicalSystem

class UnifiedChemicalSystem(*system_block: str*)

A class representing a chemical system in the Amsterdam Modeling Suite.

property adfprops: UnifiedADFPropertiesList | None

An optional list of all ADFProperties for all atoms.

property bandprops: UnifiedBANDPropertiesList | None

An optional list of all BANDProperties for all atoms.

property qeprops: UnifiedQEPropertiesList | None

An optional list of all QEProperties for all atoms.

property dftbprops: UnifiedDFTBPropertiesList | None

An optional list of all DFTBProperties for all atoms.

property reaxffprops: UnifiedReaxFFPropertiesList | None

An optional list of all ReaxFFProperties for all atoms.

property forcefieldprops: UnifiedForcefieldPropertiesList | None

An optional list of all ForcefieldProperties for all atoms.

property guiprops: UnifiedGUIPropertiesList | None

An optional list of all GUIProperties for all atoms.

enable_atomic_properties(*group_prefix: Literal['gui', 'adf', 'band', 'forcefield', 'dftb', 'reaxff', 'qe']*) → None

Enables the use of a group of atomic properties.

disable_atomic_properties(*group_prefix: Literal['gui', 'adf', 'band', 'forcefield', 'dftb', 'reaxff', 'qe']*) → None

Disables the use of a group of atomic properties. Any set properties within the group will be discarded.

atomic_properties_enabled(*group_prefix: Literal['gui', 'adf', 'band', 'forcefield', 'dftb', 'reaxff', 'qe']*) → bool

Checks if a group of atomic properties is enabled or not.

has_ghost_atoms() → bool

Checks whether the ChemicalSystem contains any Ghost atoms.

Ghost atoms are a special construct in AMS for calculating basis set superposition errors. Please refer to the ADF and BAND manuals for details.

Properties within the Atom class:

class UnifiedAtom

class UnifiedAtom(*Z: int*)

class UnifiedAtom(*element:* UnifiedElement *(page 98)*)

class UnifiedAtom(*symbol: T_AtomSymbol*)

Class representing a single atom and its properties.

property mass: float

The mass of the atom in dalton.

property is_ghost: bool

Whether the atom is a Ghost atom.

Ghost atoms are a special construct in AMS for calculating basis set superposition errors. Please refer to the ADF and BAND manuals for details.

property adf: *UnifiedADFProperties* (page 43) | None

Atomic properties used by the ADF engine.

property band: *UnifiedBANDProperties* (page 45) | None

Atomic properties used by the AMS BAND engine.

property qe: UnifiedQEProperties | None

Atomic properties used by the AMS QuantumESPRESSO engine.

property dftb: *UnifiedDFTBProperties* (page 45) | None

Atomic properties used by the AMS DFTB engine.

property reaxff: *UnifiedReaxFFProperties* (page 45) | None

Atomic properties used by the AMS ReaxFF engine.

property forcefield: *UnifiedForcefieldProperties* (page 45) | None

Atomic properties used by the AMS Forcefield engine.

property gui: *UnifiedGUIProperties* (page 46) | None

Atomic properties used for visualization in the GUI.

Classes for the Engine/Module-Specific Properties:

class UnifiedADFProperties

Atomic properties used by the ADF engine.

clear() → None

Unsets any ADF atomic properties.

copy() → *UnifiedADFProperties* (page 43)

Creates a deep copy of the ADFProperties.

empty() → bool

Returns whether any of the ADF atomic properties is set.

property ChgU: float | None

Optional atom centered point charge for ADF's 3D-RISM method.

property EpsU: float | None

Lennard-Jones parameter for ADF's 3D-RISM method.

property R: str | None

The radius of the atom for constructing the COSMO surface.

If the string represents a real number that is used as the radius, otherwise the string is used for looking up the radius in the `Solvation%Radii` block.

property SigU: float | None

Lennard-Jones parameter for ADF's 3D-RISM method.

property f: str | None

Specifies which fragment an atom belongs to; see the Fragments block in the ADF engine input.

property fg: str | None

TODO

property fs: str | None

TODO

property imol: str | None

TODO

property nuclear_charge: float | None

Alternative nuclear charge for an atom.

property type: str | None

The atom sub-type to be used by the ADF engine.

This can be used to make the ADF engine treat two atoms of the same elements as separate atom types:

```
System
    Atoms
        C  x y z   adf.type=1
        C  x y z   adf.type=2
    End
End
```

This is used as a replacements for the "dot-notation" from ADF<2025, which is no longer allowed in ADF>=2025:

```
System
    Atoms
        C.1  x y z
        C.2  x y z
    End
End
```

property x: ndarray[Any, dtype[float64]] | None

Custom atomic x-axis to be used by the ADF engine.

property z: ndarray[Any, dtype[float64]] | None

Custom atomic z-axis to be used by the ADF engine.

class UnifiedBANDProperties

Atomic properties used by the AMS BAND engine.

clear() → None

Unsets any BAND atomic properties.

copy() → *UnifiedBANDProperties* (page 45)

Creates a deep copy of the BANDProperties.

empty() → bool

Returns whether any of the BAND atomic properties is set.

property nuclear_charge: float | None

Alternative nuclear charge for an atom.

class UnifiedDFTBProperties

Atomic properties used by the AMS DFTB engine.

clear() → None

Unsets any DFTB atomic properties.

copy() → *UnifiedDFTBProperties* (page 45)

Creates a deep copy of the DFTBProperties.

empty() → bool

Returns whether any of the DFTB atomic properties is set.

property Vext: float | None

The external potential at the atomic nucleus used by the DFTB engine.

class UnifiedReaxFFProperties

Atomic properties used by the AMS ReaxFF engine.

clear() → None

Unsets any ReaxFF atomic properties.

copy() → *UnifiedReaxFFProperties* (page 45)

Creates a deep copy of the ReaxFFProperties.

empty() → bool

Returns whether any of the ReaxFF atomic properties is set.

property charge: float | None

The charge of an atom used for electrostatics by the ReaxFF engine.

class UnifiedForcefieldProperties

Atomic properties used by the AMS Forcefield engine.

clear() → None

Unsets any Forcefield atomic properties.

copy() → *UnifiedForcefieldProperties* (page 45)

Creates a deep copy of the ForcefieldProperties.

empty() → bool

Returns whether any of the Forcefield atomic properties is set.

property charge: float | None

The charge of an atom used for electrostatics by the Forcefield engine.

property type: str | None

The atom type to be used by the Forcefield engine.

class UnifiedGUIProperties

Atomic properties used for visualization in the GUI.

clear() → None

Unsets any GUI atomic properties.

copy() → *UnifiedGUIProperties* (page 46)

Creates a deep copy of the GUIProperties.

empty() → bool

Returns whether any of the GUI atomic properties is set.

property color: Tuple[int, int, int] | None

The color of an atom.

property radius: float | None

The radius of an atom for visualization purposes.

4.1.6 Molecular properties

These are properties of the Chemical System:

class UnifiedChemicalSystem

class UnifiedChemicalSystem(*system_block: str*)

A class representing a chemical system in the Amsterdam Modeling Suite.

property charge: float

The total charge of the system in atomic units (i.e. in units of elementary charge).

total_mass() → float

Total mass in atomic mass units (dalton).

formula() → str

Chemical formula in Hill notation (e.g. H2O, CH4):

- If Carbon is present, the order is: C, H, alphabetical.
- If Carbon is not present all the elements (including Hydrogen) are listed alphabetically.
- Single-letter elements come before 2-letters elements (e.g. 'K' comes before 'Kr', 'B' before 'Be')

4.1.7 Geometry and manipulation

The Chemical System provides a range of methods for manipulating the geometry and retrieving geometrical properties of your molecular system.

Here is a simple example showcasing some of the methods:

```
from scm.libbase import UnifiedChemicalSystem as ChemicalSystem

# Initialize a Chemical System from an XYZ file
mol = ChemicalSystem.from_xyz("some_mol.xyz")
print("Original system:")
print(mol)

# Translate the system to center its geometric center at the origin
mol.translate(-mol.geometric_center())

# Define a rotation matrix for a 90-degree rotation around the z-axis
rot_mat = [[0,-1, 0],
           [1, 0, 0],
           [0, 0, 1]]

# Apply the rotation
mol.rotate(rot_mat)
print("Roto-translated system:")
print(mol)
```

This will output:

```
Original system:
System
    Atoms
        H     0.0000000000000000  0.0000000000000000  0.0000000000000000
        F     1.0000000000000000  0.0000000000000000  0.0000000000000000
    End
End

Roto-translated system:
System
    Atoms
        H     0.0000000000000000 -0.5000000000000000  0.0000000000000000
        F     0.0000000000000000  0.5000000000000000  0.0000000000000000
    End
End
```

Manipulation in internal coordinates

class InternalCoordinateManipulationPolicy(*value*)

Enum representing the options for how atoms move when changes are made in internal coordinates.

Values:

- FirstPartMoves:
- SecondPartMoves:
- LighterPartMoves:
- HeavierPartMoves:

Methods for changing distances between atoms:

class UnifiedChemicalSystem

class UnifiedChemicalSystem(*system_block: str*)

A class representing a chemical system in the Amsterdam Modeling Suite.

get_distance(*a: int*, *b: int*, *unit: str = 'angstrom'*) → float

get_distance(*a: UnifiedAtom*, *b: UnifiedAtom*, *unit: str = 'angstrom'*) → float

Measures the distance between two atoms.

set_distance(*a: int*, *b: int*, *dist: float*, *policy:* InternalCoordinateManipulationPolicy *(page 47) = InternalCoordinateManipulationPolicy.LighterPartMoves*) → None

set_distance(*a: int*, *b: int*, *dist: float*, *unit: str*, *policy:* InternalCoordinateManipulationPolicy *(page 47) = InternalCoordinateManipulationPolicy.LighterPartMoves*) → None

set_distance(*a: int*, *b: int*, *dist: float*, *moving_atoms: ArrayLike*) → None

set_distance(*a: int*, *b: int*, *dist: float*, *unit: str*, *moving_atoms: ArrayLike*) → None

Sets the distance between two atoms.

moving_atoms_for_distance_change(*a: int*, *b: int*, *policy:* InternalCoordinateManipulationPolicy *(page 47) = InternalCoordinateManipulationPolicy.LighterPartMoves*) → NDArray[np.int_]

Determines which atoms will move when changing the distance between two atoms.

Methods for changing angles:

class UnifiedChemicalSystem

class UnifiedChemicalSystem(*system_block: str*)

A class representing a chemical system in the Amsterdam Modeling Suite.

set_angle(*a: int*, *b: int*, *c: int*, *angle: float*, *policy:* InternalCoordinateManipulationPolicy *(page 47) = InternalCoordinateManipulationPolicy.SecondPartMoves*) → None

set_angle(*a: int*, *b: int*, *c: int*, *angle: float*, *unit: str*, *policy:* InternalCoordinateManipulationPolicy *(page 47) = InternalCoordinateManipulationPolicy.SecondPartMoves*) → None

set_angle(*a: int*, *b: int*, *c: int*, *angle: float*, *rotation_axis: ArrayLike*, *moving_atoms: ArrayLike*) → None

set_angle(*a: int*, *b: int*, *c: int*, *angle: float*, *unit: str*, *rotation_axis: ArrayLike*, *moving_atoms: ArrayLike*) → None

Sets the angle between three atoms.

moving_atoms_for_angle_change(*a: int*, *b: int*, *c: int*, *policy:* InternalCoordinateManipulationPolicy *(page 47) = InternalCoordinateManipulationPolicy.SecondPartMoves*) → NDArray[np.int_]

Determines which atoms will move when changing the angle between three atoms.

rotation_axis_for_angle_change(*a: int*, *b: int*, *c: int*) → ndarray[Any, dtype[float64]]

Determines the rotation axis for changing the angle between three atoms.

Methods for changing dihedral angles:

class UnifiedChemicalSystem

class UnifiedChemicalSystem(*system_block: str*)

A class representing a chemical system in the Amsterdam Modeling Suite.

set_dihedral(*a: int*, *b: int*, *c: int*, *d: int*, *angle: float*, *policy:* InternalCoordinateManipulationPolicy *(page 47) = InternalCoordinateManipulationPolicy.SecondPartMoves*) → None

set_dihedral (*a: int*, *b: int*, *c: int*, *d: int*, *angle: float*, *unit: str*, *policy:* InternalCoordinateManipulationPolicy *(page 47)* = *InternalCoordinateManipulationPolicy.SecondPartMoves*) → None

set_dihedral (*a: int*, *b: int*, *c: int*, *d: int*, *angle: float*, *moving_atoms: ArrayLike*) → None

set_dihedral (*a: int*, *b: int*, *c: int*, *d: int*, *angle: float*, *unit: str*, *moving_atoms: ArrayLike*) → None

Sets the dihedral between four atoms.

moving_atoms_for_dihedral_change (*a: int*, *b: int*, *c: int*, *d: int*, *policy:* InternalCoordinateManipulationPolicy *(page 47)* = *InternalCoordinateManipulationPolicy.SecondPartMoves*) → NDArray[np.int_]

Determines which atoms will move when changing the dihedral between four atoms.

Other methods

class UnifiedChemicalSystem

class UnifiedChemicalSystem (*system_block: str*)

A class representing a chemical system in the Amsterdam Modeling Suite.

translate (*shift: ArrayLike*, *unit: str = 'angstrom'*) → None

Translates all atoms in the ChemicalSystem by a vector.

rotate (*rot_mat: ArrayLike*) → None

Rotate the system according to the rotation matrix.

align_to (*other:* UnifiedChemicalSystem *(page 83)*) → None

Translate and rotate the system to maximally align it with 'other'. It will first translate the system to the center of mass of 'other', then it will use the Kabsch algorithm (https://en.wikipedia.org/wiki/Kabsch_algorithm) to rotate the system in order to minimize the RMSD.

Notes:

- The two UnifiedChemicalSystems must have the same number of atoms.
- The atoms in the two chemical system must be in the same order.

classmethod rotation_matrix_minimizing_rmsd (*a:* UnifiedChemicalSystem *(page 83)*, *b:* UnifiedChemicalSystem *(page 83)*) → ndarray[Any, dtype[float64]]

Given two chemical systems, returns the rotation matrix that minimizes the RMSD.

classmethod rmsd (*a:* UnifiedChemicalSystem *(page 83)*, *b:* UnifiedChemicalSystem *(page 83)*, *align: bool*, *unit: str = 'angstrom'*) → float

Computes the RMSD between two systems. 'align': whether or not the systems should be roto-translated as to minimize the RMSD.

Notes:

- The two UnifiedChemicalSystems must have the same number of atoms.
- The atoms in the two chemical system must be in the same order.

geometric_center (*unit: str = 'angstrom'*) → ndarray[Any, dtype[float64]]

Position of the geometric center.

center_of_mass (*unit: str = 'angstrom'*) → ndarray[Any, dtype[float64]]

Position of the center of mass.

bounding_box_volume(*unit: str = 'angstrom3'*) → float

Volume of the bounding box.

For periodic systems, the entire extend of the lattice is used in the periodic directions. Example: for a slab (in the xy-plane), the bounding box volume is the cell's area times the extent of the atomic coordinates in the z-direction.

inertia_tensor() → ndarray[Any, dtype[float64]]

Returns the system's inertia tensor as a 3x3 matrix (units amu*angstrom^2).

The inertia tensor is not defined for periodic systems. Calling this method on a periodic systems throws a ChemicalSystemError.

moments_of_inertia() → Tuple[ndarray[Any, dtype[float64]], ndarray[Any, dtype[float64]]]

Calculates the system's moments of inertia and the corresponding principal axes.

Returns a tuple:

- a 3 component np.array of the inertial moments (units amu*angstrom^2) sorted by magnitude
- a 3x3 np.array with the corresponding axes as the column vectors

The moments of inertia and the principal axes are not defined for periodic systems. Calling this method on a periodic systems throws a ChemicalSystemError.

is_linear() → bool

Checks if a ChemicalSystem is linear.

A molecule is considered linear if all of its atoms are located on a line. A 1D periodic system is considered linear if all atoms are located on a line parallel to the only lattice vector. Slab and bulk periodic systems can never be linear.

perturb_coordinates(*noise_level: float*, *unit: str = 'angstrom'*) → None

Perturb the atomic coordinates by adding random numbers between [-noise_level,noise_level] to each Cartesian component.

This can be useful if you want to break the symmetry of your system (e.g. for a geometry optimization).

symmetrize_molecule(*tolerance: float = 0.05*) → T_MoleculeSchoenfliesSymbols

Symmetrizes and reorients a molecule. Returns the Schoenflies symbol of the point group to which the system was symmetrized.

Symmetrizes the atomic coordinates to machine precision and rototranslates the molecule to the AMS standard orientation. This allows optimal use of the system's symmetry for calculations with the AMS driver and its engines. The standard orientation has the following properties:

- The origin is a fixed point of the symmetry group.
- The z-axis is the main rotation axis.
- The xy-plane is the sigma_h plane (axial groups, C(s)).
- The x-axis is a C_2 axis (D symmetries).
- The xz-plane is a sigma_v plane (C_nv symmetries).
- In T_d and O_h the z-axis is a fourfold axis (S_4 and C_4, respectively) and the (111)-direction is a threefold axis.

See also the Symmetry section in the AMS driver manual's appendix for more information about usage of molecular symmetry in AMS.

Calling this method may be useful if the system is almost symmetric or to rototranslate a symmetric molecule into the AMS standard orientation.

Note that at the moment it is not possible to symmetrize systems with more than 2000 atoms. Calling this method with a periodic system throws an exception. See the equivalent `symmetrize_cell` method for working with periodic structures.

symmetrize() → str

Symmetrizes a system, using either `symmetrize_molecule` or `symmetrize_cell`, depending on periodicity. Returns the Schoenflies symbol of the group to which the system was symmetrized.

check_molecule_symmetry(*label: T_MoleculeSchoenfliesSymbols | str*, *tolerance: float = 1e-07*) → bool

Checks if a molecule has a particular symmetry given by a Schoenflies symbol.

Also checks that the molecule is in the AMS standard orientation for the specified point group. See the description of the `symmetrize_molecule` method for information about the AMS standard orientation.

The main use-case for this method is to check if a molecule is symmetric and oriented such that AMS can make full use of its symmetry.

4.1.8 Bonds

The Chemical System contains a `bonds` property, which is an instance of the `UnifiedBonds` class. This contains the (possibly empty) bonding information on the system.

Creating and adding bonds

There are several ways of getting bonds into your chemical system. The following snippet will show some of the most common approaches:

```
from scm.libbase import UnifiedChemicalSystem as ChemicalSystem
from scm.libbase import UnifiedBond as Bond

# Here we initialize a Chemical System from an XYZ file.
# The chemical system will not have any bonds at the moment, because
# bonds are not defined in the .xyz file.
water = ChemicalSystem.from_xyz("water.xyz")

# You can let the bond guessing algorithm simply guess the bonds:
water.guess_bonds()


# Or you can add them manually.
# First we clear the bonds we just guessed:
water.bonds.clear_bonds()

# and add bonds manually:
water.bonds.add_bond(0, 1, Bond(1.0)) # From the oxygen to the first Hydrogen
water.bonds.add_bond(0, 2, Bond(1.0)) # From the oxygen to the second Hydrogen


# You can also explicitly define the bonds in the System Block...
system_block = """
System
    Atoms
        O    0.0000000000000000  0.0000000000000000  0.3695041700000000
        H    0.0000000000000000  0.7838367199999999 -0.1847520900000000
        H    0.0000000000000000 -0.7838367199999999 -0.1847520900000000
    End
    BondOrders
        1 2 1.00
```

(continues on next page)

(continued from previous page)

```
        1 3 1.00
    End
End
"""

# ...and then load the system:
water = ChemicalSystem(system_block)

# This is how you can iterate over the bonds and get the indices of the bonded atoms:
for i, j, bond in water.bonds:
    print(f"Atoms: {i}-{j} Bond order: {bond.order}")
```

Note on atom indexing: as you can also see from the example above, atom indexing from within python and in the System Block differ: In python the first atom has index 0, while in the System Block the first atom has index 1.

The Bond Class

This contains the properties of a single bond.

class UnifiedBond

class UnifiedBond(*order: float*)

class UnifiedBond(*order: float*, *lattice_displacements: ArrayLike*)

A class representing a bond between two atoms.

get_inverted() → *UnifiedBond* (page 52)

Returns a copy of the bond with inverted lattice displacements.

property lattice_displacements: ndarray[Any, dtype[int64]]

Lattice displacements for the second atom that is part of the bond.

If atom A and B are bonded with lattice displacements [1, 0, 0], this means that atom A is bonded to the image of atom B that is displaced along the first lattice vector. Note that A > B in terms of atom indices.

property order: float

The order of the bond: 1 = single bond, 1.5 = aromatic bond, 2 = double bond, 3 = triple bond.

The Bonds Class

Methods to inspect the bonds:

class UnifiedBonds(*num_atoms: int*, *num_lattice_vectors: int*)

A class representing a set of bonds between atoms in a ChemicalSystem.

num_bonds() → int

num_bonds(*atom: int*) → int

num_bonds(*from_atom: int*, *to_atom: int*) → int

If called without an atom index, returns the total number of bonds between all atoms. If called with an atom index, returns the number of bonds of that atom. if called with two atom indices, returns the number of bonds between the two atoms.

atoms_are_bonded(*from_atom: int*, *to_atom: int*) → bool

Checks whether two atoms are bonded or not.

any_bond_between(*from_atoms: List[int]*, *to_atoms: List[int]*) → bool

Checks if there is any direct bond between two groups of atoms.

get_bonded_atoms (*atom: int*) → List[int]

Returns a list of unique atom indices that are bonded to the given atom index.

property num_atoms: int

The total number of atoms in the system.

property num_lattice_vectors: int

The number of lattice vectors of the system.

Methods to iterate over bonds:

class UnifiedBonds (*num_atoms: int*, *num_lattice_vectors: int*)

A class representing a set of bonds between atoms in a ChemicalSystem.

__len__ () → int

Returns the total number of bonds between all atoms.

__iter__ () → Iterator[Tuple[int, int, *UnifiedBond* (page 52)]]

Returns an iterator over all bonds.

Unpacking the iterator yields a 3-tuple of two atom indices A and B, and a Bond instance. Note that A <= B to avoid double counting.

get_bonds_for_atom (*from_atom: int*) → Iterator[Tuple[int, int, *UnifiedBond* (page 52)]]

Returns an iterator over all bonds from a particular atom given by its index.

Unpacking the iterator yields a 3-tuple of two atom indices A and B, and a Bond instance. Note that A <= B.

get_bonds_between_atoms (*from_atom: int*, *to_atom: int*) → Iterator[Tuple[int, int, *UnifiedBond* (page 52)]]

Returns an iterator over all bonds from between two atoms given by their indices.

Unpacking the iterator yields a 3-tuple of two atom indices A and B, and a Bond instance. Note that A <= B.

Methods to modify the bonds:

class UnifiedBonds (*num_atoms: int*, *num_lattice_vectors: int*)

A class representing a set of bonds between atoms in a ChemicalSystem.

add_bond (*from_atom: int*, *to_atom: int*, *bond:* UnifiedBond *(page 52)*) → None

Adds a bond between two atoms, given their indices.

add_bonds (*from_atoms: ArrayLike*, *to_atoms: ArrayLike*, *bonds: List[*UnifiedBond *(page 52)]*) → None

add_bonds (*from_atoms: ArrayLike*, *to_atoms: ArrayLike*, *orders: ArrayLike*) → None

Adds bonds between multiple atom pairs at the same time. All arguments should have the same length.

remove_bond (*bidx: int* | UnifiedBond *(page 52)*) → None

Removes a bond, given either its index in the .bonds attribute or an instance of a Bond.

remove_bonds_to_atom (*atom: int*) → None

Removes all bonds to a particular atom, given its index.

remove_bonds_to_atoms (*remove_atoms: ArrayLike*) → None

Removes all bonds for a list of atoms given by their indices.

remove_bonds_between_atoms (*from_atom: int*, *to_atom: int*) → None

Removes the bonds between two atoms, gived their indices.

clear_bonds() → None

Removes all bonds.

Direct read-only access to the modified Compressed Sparse Row format:

class UnifiedBonds(*num_atoms: int*, *num_lattice_vectors: int*)

A class representing a set of bonds between atoms in a ChemicalSystem.

property row_offset: ndarray[Any, dtype[int64]]

Bonds are stored as sparse matrix in a modified Compressed Spare Row format: The index in row_offset represents from_atom, and the value is an offset in column_index and bonds. The values in column_index represent to_atom. Because there can be duplicate indices (periodic bonds), it is not strictly a CSR matrix.

property column_index: ndarray[Any, dtype[int64]]

Bonds are stored as sparse matrix in a modified Compressed Spare Row format: The index in row_offset represents from_atom, and the value is an offset in column_index and bonds. The values in column_index represent to_atom. Because there can be duplicate indices (periodic bonds), it is not strictly a CSR matrix.

property bond_index: ndarray[Any, dtype[int64]]

Bonds are stored as sparse matrix in a modified Compressed Spare Row format: The index in row_offset represents from_atom, and the value is an offset in column_index and bonds. The values in column_index represent to_atom. Because there can be duplicate indices (periodic bonds), it is not strictly a CSR matrix.

property bonds: UnifiedBondList

Bonds are stored as sparse matrix in a modified Compressed Spare Row format: The index in row_offset represents from_atom, and the value is an offset in column_index and bonds. The values in column_index represent to_atom. Because there can be duplicate indices (periodic bonds), it is not strictly a CSR matrix.

Supported file formats and conversions to other formats:

class UnifiedBonds(*num_atoms: int*, *num_lattice_vectors: int*)

A class representing a set of bonds between atoms in a ChemicalSystem.

classmethod from_kf(*kf:* KFFile *(page 80)*, *section: str*) → UnifiedBonds

Constructs and returns new Bonds from a section on a KF file.

write_kf(*kf:* KFFile *(page 80)*, *section: str*, *write_list_format: bool = True*) → None

Writes Bonds to a section on a KF file.

The `write_list_format` argument determines whether the retrocompatible list format (with variables `fromAtoms`, `toAtoms`, `bondOrders`) is written. Otherwise the Compressed Sparse Row format is written directly.

classmethod from_list(*num_atoms: int*, *num_lattice_vectors: int*, *from_atoms: ArrayLike*, *to_atoms: ArrayLike*, *bond_orders: ArrayLike*, *lattice_displacements: ArrayLike = [[]]*, *indexing: int = 0*) → UnifiedBonds

Creates a set of bonds from lists of atom indices, bond orders and optional lattice displacements.

The `indexing` argument can be used to switch to one-based atom indices, as they are used in the AMS System block format.

classmethod from_sparse(*num_lattice_vectors: int*, *row_offset: ArrayLike*, *column_index: ArrayLike*, *bond_orders: ArrayLike*, *lattice_displacements: ArrayLike = [[]]*, *indexing: int = 0*) → UnifiedBonds

Low level construction of a set of bonds directly from the (modified) CRS storage.

The `indexing` argument can be used to switch to one-based atom indices, as they are used in the AMS System block format.

Methods in the Chemical System

class UnifiedChemicalSystem

class UnifiedChemicalSystem(*system_block: str*)

A class representing a chemical system in the Amsterdam Modeling Suite.

property bonds: UnifiedBonds

The bonds of the system.

has_bonds() → bool

Checks if the system has any bonding information.

guess_bonds() → None

Guesses bonds based on the atomic elements and the geometry. Keeps existing bonds.

set_lattice_displacements_from_minimum_image_convention() → None

Applies the minimum image convention which sets the lattice_displacements for all bonds to be the shortest possible.

num_molecules() → int

Counts the number of connected molecules that are part of this system.

What is considered a molecule is based on the bonds of the system: anything connected by any bonds is considered to be a molecule. For a fully connected system, this method returns `1`. If the system has no bonding information, each atom is its own molecule and this method returns the total number of atoms in the system.

molecule_indices() → ndarray[Any, dtype[int64]]

Returns a num_atoms sized array mapping the atoms to connected molecules of the system.

What is considered a molecule is based on the bonds of the system: anything connected by any bonds is considered to be a molecule. For a system consisting of two water molecules (where the first three atoms make up one of the molecules), this method returns `[0, 0, 0, 1, 1, 1]`. If the system has no bonding information, each atoms is considered its own molecule and this method returns `[0, 1, 2, ..., num_atoms]`.

split_into_molecules() → List[*UnifiedChemicalSystem* (page 83)]

Splits the system into individual molecules based on the connectivity between atoms.

The length of the returned list is `num_molecules`. Which set of atoms ends up in which element of the list is determined by the indices returned by the `molecule_indices` method.

The returned systems will have the same lattice as the original and a total charge of zero. Atomic properties and regions are preserved, but the returned systems do not have any selected atoms.

This method is just a shortcut for: `cs.split(cs.molecule_indices())`.

bond_cuts_molecule(*from_atom: int*, *to_atom: int*) → bool

bond_cuts_molecule(*from_atom: UnifiedAtom*, *to_atom: UnifiedAtom*) → bool

Checks if removing the bonds between two atoms would cut the graph into two disjoint subgraphs.

atom_is_in_ring(*atom: int | UnifiedAtom*) → bool

Checks if an atom is part of any ring.

An atom is in a ring, if removing the bond to (at least) one of its neighbors does not cut the graph into two disjoint subgraphs.

molgraph_dijkstra (*from_atidx: int*, *dist_func: molgraph_edgeweight_func_t*, *to_atidx: int = -1*) → Tuple[List[float], List[int], List[bool]]

General method implementing the Dijkstra algorithm on the molecular graph. The Dijktra algorithm solves the single-source shortest path problem in weighted graphs with non-negative weights.

- `from_atidx` is the index of source atom, i.e. the atom from which we start searching for paths.
- `dist_func` is a method (of type `molgraph_edgeweight_func_t = Callable[[int, int, UnifiedBond], float]`) taking two atomic indices and a Bond instance as input. It returns the weight of the graph edge, aka the distance between the two atoms. The returned distance may not be negative, but you can return `float("inf")` to ignore a particular edge in the search.
- `to_atidx` marks an optional target atom and can be used to solve the single-pair shortest path problem by prematurely terminating the Dijkstra algorithm as soon as the shortest path to the target atom has been found.

Returns a 3-tuple consisting of:

1. `path_lengths` is a list of length `num_atoms` that upon return of the method will be populated with the lengths of the shortest paths from the source atom. Unreachable atoms, aka atoms in other molecules, will have an infinite path length.
2. `path_through` is a list of length `num_atoms` that upon return will for each atom contain the preceding atoms index in the shortest path from source atom. By stepping backwards through this array all shortest paths from the source atom can be recovered. If no path has been found to a particular atom, the corresponding list element will be set to `-1`.
3. `visited` is a list of length `num_atoms` marking which atoms have been visited by the Dijkstra algorithm prior to its termination. An atom counts as visited if it has been popped of the top of the priority queue to have its neighbors added to the queue. Note that in case of premature termination of the algorithm due to using the `to_atidx` argument, shortest paths have only been determined for all visited atoms. (Other atoms may still have a path length set, but it is not guaranteed to be the shortest path from the source.)

shortest_path_lengths_from (*from_atom: int | UnifiedAtom*) → List[float]

shortest_path_lengths_from (*from_atom: int | UnifiedAtom*, *dist_func: molgraph_edgeweight_func_t*) → List[float]

Returns the lengths of all shortest paths from `from_atom` to all other atoms.

shortest_path_between (*from_atom: int*, *to_atom: int*, *dist_func: molgraph_edgeweight_func_t*) → List[int] | None

shortest_path_between (*from_atom: int*, *to_atom: int*) → List[int] | None

shortest_path_between (*from_atom: UnifiedAtom*, *to_atom: UnifiedAtom*, *dist_func: molgraph_edgeweight_func_t*) → List[int] | None

shortest_path_between (*from_atom: UnifiedAtom*, *to_atom: UnifiedAtom*) → List[int] | None

Returns the shortest path between two atoms as a list of atom indices.

Note that the list contains both `from_atom` and `to_atom` as the first and last element. The length of the path (measured in number of hops) is therefore equal to the length of the returned list minus one.

shortest_path_length_between (*from_atom: int*, *to_atom: int*, *dist_func: molgraph_edgeweight_func_t*) → float

shortest_path_length_between (*from_atom: int*, *to_atom: int*) → float

shortest_path_length_between (*from_atom: UnifiedAtom*, *to_atom: UnifiedAtom*, *dist_func: molgraph_edgeweight_func_t*) → float

shortest_path_length_between (*from_atom: UnifiedAtom*, *to_atom: UnifiedAtom*) → float

Returns the length of the shortest path between two atoms.

4.1.9 Lattice and Periodic Systems

The Chemical System can handle periodic system with arbitrary numbers of Periodic Boundaries Conditions (i.e. 0,1,2,3).

The `lattice` property of the Chemical System contains the lattice vectors information.

class UnifiedChemicalSystem

class UnifiedChemicalSystem(*system_block: str*)

A class representing a chemical system in the Amsterdam Modeling Suite.

property lattice: *UnifiedLattice* (page 57)

The lattice of the system.

has_lattice() → bool

Checks if the system has a lattice.

Note that the `lattice` attribute of a ChemicalSystem is never None. A system is considered to not have a lattice if the number of lattice vectors is 0. Using this method is the same a checking `lattice.num_vectors == 0`.

Here are some examples:

```
from scm.libbase import UnifiedChemicalSystem as ChemicalSystem
from scm.libbase import UnifiedLattice as Lattice

# Create an empty chemical system
chain = ChemicalSystem()

# Add an Helium atom in the origin
chain.add_atom("He", [0, 0, 0])

# Make it a 1D periodic chain by adding a lattice vector along X (units: Bohr)
chain.lattice = Lattice([[2, 0, 0]])
print(chain)

# You can also initialize it from a System Block:
graphene = ChemicalSystem("""
System
     Atoms
          C    0.0000000000000000  0.0000000000000000  0.0000000000000000
          C    1.2300000000000000  0.7101408311032394  0.0000000000000000
     End
     Lattice
           2.4600000000000000   0.0000000000000000   0.0000000000000000
          -1.2300000000000000   2.1304224933097191   0.0000000000000000
     End
End
""")

print(graphene)
```

These are the methods in the UnifiedLattice class.

class UnifiedLattice

class UnifiedLattice(*vectors: ArrayLike*, *unit: str = 'angstrom'*)

A class representing the lattice of a ChemicalSystem.

cartesian_to_fractional(*cartesian_coord: ArrayLike*, *unit: str = 'angstrom'*) → ndarray[Any, dtype[float64]]

Convert Cartesian coordinates to fractional coordinates for a single point. No conversion will take place in non-periodic directions.

cartesians_to_fractionals (*cartesian_coords: ArrayLike*, *unit: str = 'angstrom'*) → ndarray[Any, dtype[float64]]

Convert Cartesian coordinates to fractional coordinates for a set of points. No conversion will take place in non-periodic directions.

copy () → *UnifiedLattice* (page 57)

Creates a deep copy of the Lattice.

fractional_to_cartesian (*fractional_coord: ArrayLike*, *unit: str = 'angstrom'*) → ndarray[Any, dtype[float64]]

Convert fractional coordinates to Cartesian coordinates for a single point. No conversion will take place in non-periodic directions.

fractionals_to_cartesians (*fractional_coords: ArrayLike*, *unit: str = 'angstrom'*) → ndarray[Any, dtype[float64]]

Convert fractional coordinates to Cartesian coordinates for a set of points. No conversion will take place in non-periodic directions.

classmethod from_kf (*kf:* KFFile *(page 80)*, *section: str*) → *UnifiedLattice* (page 57)

Constructs and returns a new Lattice from a section on a KF file.

get_angles (*unit: str = 'rad'*) → ndarray[Any, dtype[float64]]

Get the angles between the lattice vectors (note: always returns 3 numbers).

get_cell_depth (*unit: str = 'angstrom'*) → ndarray[Any, dtype[float64]]

Return unit cell depth, or thickness, in each dimension.

The cell depth is the distance between the two opposite bounding planes of the parallelepiped spanned by the lattice vectors. For three lattice vectors `v_i`, `v_j`, and `v_k`, the `i`-th component of the cell depth is the height of lattice vector i over the plane spanned by `v_j` and `v_k`.

Always returns a 3-component vector. For < 3D periodic systems, the remaining components for the non-periodic directions at the end are set to 0.

get_lengths (*unit: str = 'angstrom'*) → ndarray[Any, dtype[float64]]

Get the length of the lattice vectors (note: always returns 3 numbers).

get_reciprocal_lattice_vectors (*unit: str = 'angstrom-1'*) → ndarray[Any, dtype[float64]]

Get the reciprocal lattice vectors as a numpy array.

get_volume (*unit: str = 'angstrom'*) → float

Get the volume of the unit cell (for 2D: area, for 1D length)

is_close (*other:* UnifiedLattice *(page 57)*, *tol: float = 0.001*, *unit: str = 'angstrom'*) → bool

is_close (*other:* UnifiedLattice *(page 57)*, *length_tol: float*, *angle_tol: float*, *length_unit: str = 'angstrom'*, *angle_unit: str = 'rad'*) → bool

Checks if two sets of lattice vectors are close to each other.

is_orthogonal () → bool

Is the lattice orthogonal?

map_vector_to_central_cell (*vec: ArrayLike*, *unit: str = 'angstrom'*) → ndarray[Any, dtype[float64]]

Maps a vector into the central cell, such that its fractional coordinates are in range [-0.5,+0.5). No mapping will take place in non-periodic directions.

map_vectors_to_central_cell (*vecs: ArrayLike*, *unit: str = 'angstrom'*) → ndarray[Any, dtype[float64]]

Maps a set of vectors into the central cell, such that their fractional coordinates are in range [-0.5,+0.5). No mapping will take place in non-periodic directions.

write_kf (*kf:* KFFile *(page 80)*, *section: str*) → None

Writes a Lattice to a section on a KF file.

property num_vectors: int

Returns the number of lattice vectors (i.e. the number of PBC).

property vectors: AngstromCoordsArray

The lattice vectors in angstrom.

Here are methods from the `ChemicalSystem` that relate to periodic systems:

class UnifiedChemicalSystem

class UnifiedChemicalSystem (*system_block: str*)

A class representing a chemical system in the Amsterdam Modeling Suite.

set_num_lattice_vectors (*nvec: int*) → None

Sets the number of lattice vectors.

Can only be used to reduce the number of periodic directions. Note that this may change the number of bonds in the system: bonds that have lattice displacements in the no longer periodic directions will be removed.

set_fractional_coordinates (*frac_coords: ArrayLike*, *unit: str = 'angstrom'*) → None

Sets the fractional coordinates of all atoms. In non-periodic directions the plain coordinates should be passed in the specified unit.

get_fractional_coordinates (*unit: str = 'angstrom'*) → ndarray[Any, dtype[float64]]

Returns the fractional coordinates of all atoms with respect to the cell. In non-periodic directions the plain coordinates are returned in the specified unit.

map_atoms (*start_range: float | ArrayLike*) → Tuple[bool, ndarray[Any, dtype[int64]]]

Maps all atoms into a unit cell from [start_range:start_range+1] in fractional coordinates.

No mapping will take place in non-periodic directions. The returned tuple signifies if any mapping has taken place and the shifts in units of the lattice vectors that would move the mapped atoms back to their original position.

map_atoms_around_atom (*atom: int | UnifiedAtom*) → Tuple[bool, ndarray[Any, dtype[int64]]]

Map all atoms around the chosen atom that will be in the center of the new unit cell.

No mapping will take place in non-periodic directions. The returned tuple signifies if any mapping has taken place and the shifts in units of the lattice vectors that would move the mapped atoms back to their original position.

Note that the atom around which we are mapping does not move in the process. If you want the central atom to be in a special position (e.g. the origin), you will have to shift all atoms separately using the `translate` method.

map_atoms_continuous () → bool

Map all atoms that are bonded across the periodic boundary back.

Imagine a box full of molecules, with all atoms in the range [0,1] in fractional coordinates. Molecules that cross between cells will have bonds with lattice displacements in this case. Calling this method will remove all lattice displacements and move atoms outside of the range [0,1] in fractional coordinates. After calling it, all molecules are "continuous" and extend outside of the original cell.

No mapping will take place in non-periodic directions.

The return value indicates whether the method was able to completely remove all lattice displacements. In cases where a structure is connected to itself on both ends, it is not possible to make it continuous. An example of this would be a polymer chain going through the cell and connecting to itself across two cell boundaries at the same time. In this case the return value is `False`, but disconnected parts of the system would still have been made continuous.

apply_strain (*strain_matrix: ArrayLike*) → None

Apply a strain deformation to a periodic system.

The atoms in the unit cell will be strained accordingly, keeping the fractional atomic coordinates constant.

The strain_matrix argument should be (or be convertable to) an NxN numpy array, where N is the number of lattice vectors of the system.

apply_strain_voigt (*strain_voigt: Sequence[float]*) → None

Apply a strain deformation to a periodic system.

The atoms in the unit cell will be strained accordingly, keeping the fractional atomic coordinates constant.

The strain_voigt argument should be passed in Voigt form:

- for 3D periodic systems: [e_xx, e_yy, e_zz, gamma_yz, gamma_xz, gamma_xy]
- for 2D periodic systems: [e_xx, e_yy, gamma_xy]
- for 1D periodic systems: [e_xx]

with e.g. e_xy = gamma_xy/2.

perturb_lattice (*noise_level: float*) → None

Perturb the lattice vectors by applying random strain with matrix elements between [-noise_level,noise_level].

This can be useful if you want to deviate from an ideal symmetric geometry, for example if you look for a phase change due to high pressure.

make_supercell (*supercell: ArrayLike*) → *UnifiedChemicalSystem* (page 83)

Create a supercell by scaling the lattice vectors.

Returns a new system where copies of atoms will have the same properties as in the initial unit cell. Bonds are replicated between copies and across the new unit cell boundary.

make_supercell_trafo (*supercell: ArrayLike*) → *UnifiedChemicalSystem* (page 83)

Create a supercell by creating the matrix product of the lattice vectors and the supercell matrix.

Returns a new system where copies of atoms will have the same properties as in the initial unit cell. Bonds are replicated between copies and across the new unit cell boundary.

make_slab_thickness (*ref_atom: int*, *top: float*, *bottom: float*, *miller: ArrayLike*, *translate: float = 0.0*, *unit: str = 'angstrom'*) → *UnifiedChemicalSystem* (page 83)

Create a new system which is a 2D slab, by slicing a 3D system, specifying the upper and lower bound of the resulting slab.

make_slab_layers (*ref_atom: int*, *num_layers: int*, *miller: ArrayLike*, *translate: float = 0.0*, *unit: str = 'angstrom'*) → *UnifiedChemicalSystem* (page 83)

Create a new system which is a 2D slab, by slicing a 3D system, specifying the number of layers in the resulting slab.

make_primitive_cell (*precision: float = 0.1*) → *UnifiedChemicalSystem* (page 83)

Create a new system by converting a 2D or 3D cell to its primitive representation.

make_conventional_cell (*precision: float = 0.1*) → *UnifiedChemicalSystem* (page 83)

Create a new system by converting a 2D or 3D cell to its conventional representation.

density (*unit: str = 'dalton/angstrom3'*) → float

Returns the density of the system in the specified unit. Only valid for 3D periodic systems.

set_density (*target_density: float*, *unit: str = 'dalton/angstrom3'*) → None

Applies a uniform strain to match the specified target density. Only valid for 3D periodic systems.

4.1.10 Environment and electrostatic embedding

class UnifiedChemicalSystem

class UnifiedChemicalSystem (*system_block: str*)

A class representing a chemical system in the Amsterdam Modeling Suite.

property electrostatic_embedding: *UnifiedElectrostaticEmbedding* (page 61)

The electrostatic embedding of the system.

These are the methods in the UnifiedElectrostaticEmbedding class.

class UnifiedElectrostaticEmbedding

A class containing information about electrostatic fields in which the system is embedded.

add_multipole (*r: ArrayLike*, *M: ArrayLike*, *r_unit: str = 'angstrom'*) → None

Adds a new multipole to the electrostatic embedding.

- `r` is the position of the multipole in cartesian coordinates.
- `M` are the values of the multipole up to a certain l-value, e.g. 1 number for a simple point charge, 4 numbers for a point charge with a dipole moment, or 9 numbers for a point charge with a dipole and quadrupole moment.
- `r_unit` is the length unit for the cartesian coordinates.

add_multipoles (*r: ArrayLike*, *M: ArrayLike*, *r_unit: str = 'angstrom'*) → None

Adds a set of new multipoles to the electrostatic embedding.

- `r` is the position of the multipoles in cartesian coordinates as a N x 3 array.
- `M` are the values of the multipole up to a certain l-value, as a N x M array, where M is e.g. 1 number for a simple point charge, 4 numbers for a point charge with a dipole moment, or 9 numbers for a point charge with a dipole and quadrupole moment.
- `r_unit` is the length unit for the cartesian coordinates.

clear_multipoles () → None

Removes all multipoles from the electrostatic embedding.

copy () → *UnifiedElectrostaticEmbedding* (page 61)

Creates a deep copy of the ElectrostaticEmbedding.

classmethod from_kf (*kf:* KFFile *(page 80)*, *section: str*) → *UnifiedElectrostaticEmbedding* (page 61)

Constructs and returns a new ElectrostaticEmbedding from a section on a KF file.

has_homogeneous_field () → bool

Whether a non-zero electric field is present.

has_multipoles() → bool

Whether any mutipole is present.

is_active() → bool

Whether there is any electrostatic embedding at all.

num_multipoles() → int

Number of multipole charges.

num_zlm() → int

Number of zlm for the multiple charges. 1 means charge only. 4 means charge and dipoles, etc…

write_kf(*kf:* KFFile *(page 80)*, *section: str*, *write_list_format: bool = True*) → None

Writes an ElectrostaticEmbedding to a section on a KF file.

property charge_width: float

The width parameter in a.u. in case a Gaussian charge model is chosen. A negative value means that the width will be chosen automatically.

property e_field: ndarray[Any, dtype[float64]]

A uniform electric field (x,y,z components, in atomic units).

property multipoles: ndarray[Any, dtype[float64]]

The multipoles (charge, dipoles…) in atomic units.

property use_charge_broadening: bool

Whether spherical Gaussian charge distribution is used for the charges. If false, a point charge will be used.

property xyz_multipoles: AngstromCoordsArray

The xyz position of the multipoles in angstrom.

4.1.11 Regions

Regions are 'groups' of atoms within a ChemicalSystem. Regions are used for some of the AMS driver and its engines features, but they can also be useful tools for bookkeeping and manipulations (see also the AMS documentation on Regions).

Here is a simple example showing how to create and use regions:

```
from scm.libbase import UnifiedChemicalSystem as ChemicalSystem
import numpy as np

# Create a ChemicalSystem and define two regions (one for each water molecule)
mol = ChemicalSystem(
"""
System
     Atoms
          O -2.676678  0.759085  0.370636 region=water_1
          H -3.468900  0.415690  0.814339 region=water_1
          H -3.005004  1.433129 -0.246320 region=water_1
          O  0.039085  0.303872  1.265871 region=water_2
          H -0.874303  0.483757  0.975166 region=water_2
          H  0.293563 -0.534617  0.849654 region=water_2
     End
End
"""
)
```

(continues on next page)

(continued from previous page)

```
# Translate all the atoms in the region "water_2"
for i_atom in mol.get_atoms_in_region("water_2"):
    mol.atoms[i_atom].coords += np.array([1, 0, 0])
```

Region names are case-sensitive and may not include certain characters. If a string is a valid name for a region can be checked with the `is_valid_region_name` method:

class UnifiedChemicalSystem

class UnifiedChemicalSystem(*system_block: str*)

A class representing a chemical system in the Amsterdam Modeling Suite.

classmethod is_valid_region_name(*name: str*) → bool

Checks if a string is a valid region name.

Valid region names do not contain any of the following characters:

```
, + - * / \ | $ & ^ < > ( ) [ ] { } " '
```

Valid region names do not have leading or trailing whitespace. Note that region names are case-sensitive, so "A" and "a" correspond to different regions.

There are various methods for querying region information:

class UnifiedChemicalSystem

class UnifiedChemicalSystem(*system_block: str*)

A class representing a chemical system in the Amsterdam Modeling Suite.

property num_regions: int

Returns the number of regions used in the system.

Note that there is no such thing as an empty region. Every region contains at least one atom.

property region_names: List[str]

Returns an ASCIIbetically ordered list of the names of all regions in the system.

Note that there is no such thing as an empty region. Every region contains at least one atom.

has_region(*region: str*) → bool

Checks if a region exists in the system, given its region name.

Note that region should be a valid region name, not a region expression. If you want to check if a region expression contains any atoms, use `num_atoms_in_region(...) > 0` instead.

do_regions_intersect(*regionA: str*, *regionB: str*) → bool

Checks if two regions or region expressions intersect, i.e. have at least one atom in common.

num_atoms_in_region(*region: str*) → int

Returns the number of atoms in a region or region expression.

Returns zero if the region does not exist at all, or if the region expression evaluates to an empty set.

num_atoms_outside_region(*region: str*) → int

Returns the number of atoms outside of a region or region expression.

Returns the total number of atoms in the system if the region does not exist at all, or if the region expression evaluates to an empty set.

get_atoms_in_region (*region: str*) → ndarray[Any, dtype[int64]]

Returns a sorted array of atom indices of all atoms in a region or region expression.

get_atoms_outside_region (*region: str*) → ndarray[Any, dtype[int64]]

Returns a sorted array of atom indices of all atoms outside of a region or region expression.

is_atom_in_region (*atom: int | UnifiedAtom*, *region: str*) → bool

Checks if an atom is in a region or region expression.

is_atom_outside_region (*atom: int | UnifiedAtom*, *region: str*) → bool

Checks if an atom is outside of a region or region expression.

get_regions_of_atom (*atom: int | UnifiedAtom*) → List[str]

Returns an ASCIIbetically sorted list of the names of all regions an atom is part of.

Almost all methods for querying region information also support so called region expressions in addition to plain region names. Region expressions may include region names, the parentheses `()`, as well as the operators `+` for union, `-` for set difference and `&` for set intersection. Additionally the wildcard `*` means all atoms, while `$` stands for the set of selected atoms.

```
# number of atoms that are either in region A or B (or both)
num_at_AB = mol.num_atoms_in_region("A + B")

# loop over all atoms that are not in the intersection of region A and B
for atidx in mol.get_atoms_in_region("* - (A & B)"):
    ...
```

The following method can be used to extract the used regions from a region expression:

class UnifiedChemicalSystem

class UnifiedChemicalSystem (*system_block: str*)

A class representing a chemical system in the Amsterdam Modeling Suite.

classmethod get_operands_in_region_expression (*region_expression: str*) → List[str]

Returns an ASCIIbetically ordered list of all unique operands used in a region expression.

Operands are either names of regions, or the special operands `*` representing all atoms, or `$` representing the set of selected atoms.

The following methods can be used to change the assignment of atoms to regions:

class UnifiedChemicalSystem

class UnifiedChemicalSystem (*system_block: str*)

A class representing a chemical system in the Amsterdam Modeling Suite.

set_atoms_in_region (*atom_indices: ArrayLike*, *region: str*) → None

Creates or sets an entire region given the indices of the atoms to be part of the region.

If atom_indices is empty and the region previously existed, it will effectively be deleted.

add_atoms_to_region (*atom_indices: ArrayLike*, *region: str*) → None

Adds multiple atoms to a region, given their atom indices.

add_all_atoms_to_region (*region: str*) → None

Adds all of the system's atoms to a region.

remove_atoms_from_region (*atom_indices: ArrayLike*, *region: str*) → None

Removes multiple atoms from a region, given their atom indices.

No error is raised if any of the atoms is not part of the region.

remove_region (*region: str*) → None

Removes a region. Atoms in that region will keep existing, but will not longer be part of that region.

remove_all_regions () → None

All atoms will keep existing, but will not longer be part of any region.

add_atom_to_region (*atom: int | UnifiedAtom*, *region: str*) → None

Adds an atom to a region.

remove_atom_from_region (*atom: int | UnifiedAtom*, *region: str*) → None

Removes an atom from a region.

Throws a ChemicalSystemError if the atom was not part of the region it should be removed from.

remove_atom_from_all_regions (*atom: int | UnifiedAtom*) → None

Removes an atom from all regions.

4.1.12 Atom selection

The ChemicalSystem keeps track of a set of selected atoms. In the GUI an atom is selected by simply clicking on it, and the current selection is highlighted by a cyan outline and shading.

The following methods allow inspecting the current selection:

class UnifiedChemicalSystem

class UnifiedChemicalSystem (*system_block: str*)

A class representing a chemical system in the Amsterdam Modeling Suite.

num_selected_atoms () → int

The number of currently selected atoms.

is_atom_selected (*atom: int | UnifiedAtom*) → bool

Checks if an atom is currently selected.

get_selected_atoms () → ndarray[Any, dtype[int64]]

Returns an array of indices of all selected atoms.

You can also use the `$` symbol to refer to the set of currently selected atoms in region expressions. The following bit of code loops over all selected atoms in the region named `myregion`:

```
for atidx in mol.get_atoms_in_region("$ & myregion"):
    ...
```

There are many methods to change the current selection. Almost all of them work by adding or removing atoms from the current selection, e.g. the `select_atom` method adds an atom to the current selection and is equivalent to clicking it in the GUI. The exception to this is the `set_selected_atoms` method, which completely replaces the current selection.

Note that the order in which atoms are selected is tracked for small selections. Small selections of up to 4 atoms are used in the GUI for interactive manipulation in internal coordinates using the sliders at the bottom of the molecule view. The selection order is relevant for manipulations of e.g. dihedral angles, as the dihedral angle between atoms `(3,1,2,4)` is different than between atoms `(1,2,3,4)`. For large selections >10 atoms there are no use cases in which the selection

order is relevant and for performance reasons `get_selected_atoms` always returns the indices of the selected atoms in ascending order.

Basic methods to change the current selection:

class UnifiedChemicalSystem

class UnifiedChemicalSystem(*system_block: str*)

A class representing a chemical system in the Amsterdam Modeling Suite.

select_atom(*atom: int | UnifiedAtom*) → None

Selects an atom, i.e. adds it to the current selection.

deselect_atom(*atom: int | UnifiedAtom*) → None

Deselects an atom, i.e. removes it from the current selection.

select_atoms(*atom_indices: ArrayLike*) → None

Selects multiple atoms at once, given their atom indices.

deselect_atoms(*atom_indices: ArrayLike*) → None

Deselects multiple atoms at once, given their atom indices.

set_selected_atoms(*atom_indices: ArrayLike*) → None

Sets the selection to the given atom indices. Any previous selection is cleared.

GUI style methods from AMSinput:

class UnifiedChemicalSystem

class UnifiedChemicalSystem(*system_block: str*)

A class representing a chemical system in the Amsterdam Modeling Suite.

select_all() → None

Selects all atoms.

deselect_all() → None

Deselects all atoms, or in other words: clears the current selection.

invert_selection() → None

Inverts the set of selected atoms.

All previously unselected atoms will become selected. All previously selected atoms will become unselected.

select_connected() → None

Selects all atoms bonded to the currently selected atoms.

select_molecule() → None

Using the bonds, selects entire molecules that include any currently selected atom.

This is equivalent to repeatedly calling `select_connected` until the selection stops growing.

select_region(*region: str*) → None

Selects atoms in a region or region expression.

select_atom_close_to_origin() → None

Selects the atom that is closest to the origin of the coordinate system.

select_within_radius(*radius: float*, *unit: str = 'angstrom'*) → None

Selects all atoms within a given radius of any of the currently selected atoms.

make_selection_cappable() → None

Extends the current selection but does not cross single bonds, unless they are to hydrogen atoms.

The intended use of this method is to select a suitable QM region for QM/MM calculations where one wants the QM region to be separated by single bonds from the rest of the molecule.

select_atoms_of_same_type() → None

Selects all atoms whose element is the same as of a currently selected atom.

Methods taking predicate functions:

class UnifiedChemicalSystem

class UnifiedChemicalSystem(*system_block: str*)

A class representing a chemical system in the Amsterdam Modeling Suite.

select_atoms_if(*pred: Callable[[UnifiedAtom], bool]*) → None

Selects atoms based on a predicate function.

deselect_atoms_if(*pred: Callable[[UnifiedAtom], bool]*) → None

Deselects atoms based on a predicate function.

select_connected_if(*pred: Callable[[UnifiedAtom, UnifiedAtom,* UnifiedBond *(page 52)], bool]*) → None

Selects atoms bonded to the currently selected atoms based on a predicate function.

The predicate function is called on a pair of atoms and their connecting bond. The unselected atom attached to the bond only becomes selected of the predicate function returns `True`.

select_molecule_if(*pred: Callable[[UnifiedAtom, UnifiedAtom,* UnifiedBond *(page 52)], bool]*) → None

Using the bonds and a predicate function, selects all molecules that include a currently selected atom.

This is equivalent to repeatedly calling `select_connected_if` with the same predicate until the selection stops growing.

4.1.13 Comparison of systems

Simple methods for comparing two ChemicalSystem instances:

class UnifiedChemicalSystem

class UnifiedChemicalSystem(*system_block: str*)

A class representing a chemical system in the Amsterdam Modeling Suite.

has_same_atoms(*other:* UnifiedChemicalSystem *(page 83), properties: List[Literal['gui', 'adf', 'band', 'forcefield', 'dftb', 'reaxff', 'qe']] = []*) → bool

has_same_atoms(*other:* UnifiedChemicalSystem *(page 83), comp: atom_comparator_func_t*) → bool

Checks if two systems have identical atoms in the same order.

Atoms are compared one by one using either a user-defined comparator function, or the `Atom.has_identical_properties` method with a user defined list of (optional) property groups to consider in the comparison.

has_same_coords(*other:* UnifiedChemicalSystem *(page 83), tol: float = 0.001, unit: str = 'angstrom'*) → bool

Checks if the atomic coordinates of two systems are within a threshold of each other.

This check is intended for systems that have the same number of atoms and all atoms in the same order. (You likely want to call `has_same_atoms` before calling this method.) The threshold `tol` is compared against the distance between two corresponding atoms. This ensures that the return value of this method does not depend on an overall rotation of the two systems.

has_same_geometry (*other:* UnifiedChemicalSystem *(page 83)*, *tol: float = 0.001*, *unit: str = 'angstrom'*) → bool

Checks if the atomic coordinates and lattice vectors of two systems are within a threshold of each other. This is just a shorthand for calling `has_same_coords` and `lattice.is_close()` on the two systems.

has_same_regions (*other:* UnifiedChemicalSystem *(page 83)*) → bool

Checks if two systems have identical regions, meaning region names match and each region includes the same atoms.

This only checks the indices of the atoms assigned to the different regions. It does not check if atoms with the same index are actually the same. Use `has_same_atoms` for that.

has_same_selection (*other:* UnifiedChemicalSystem *(page 83)*, *consider_selection_order: bool = False*) → bool

Checks if two systems have the same atom selection.

This only checks the indices of the selected atoms. It does not check if atoms with the same index are actually the same. Use `has_same_atoms` for that.

By default the selection order is ignored in the check and the two selections are compared as a set. This can be changed via the `consider_selection_order` argument.

4.1.14 Converting to and from PLAMS Molecules

ChemicalSystems can PLAMS' molecules can be converted into each other using the following conversion functions: `scm.utils.conversions.plams_molecule_to_chemsys` and `scm.utils.conversions.chemsys_to_plams_molecule`.

Example:

```
from scm.libbase import UnifiedChemicalSystem as ChemicalSystem
from scm.plams import *
from scm.utils.conversions import plams_molecule_to_chemsys, chemsys_to_plams_molecule

# Create a PLAMS molecule from a SMILES string:
plams_molecule = from_smiles('C-C')

# Convert from a PLAMS molecule to a ChemicalSystem:
chemical_system = plams_molecule_to_chemsys(plams_molecule)

# Convert back from ChemicalSystem to a PLAMS molecule:
plams_molecule = chemsys_to_plams_molecule(chemical_system)
```

4.1.15 Interoperability with other Python libraries

ChemicalSystem instances can be convert to and from the ase.Atoms (https://wiki.fysik.dtu.dk/ase/ase/atoms.html) class that is part of the Atomic Simulation Environment (https://wiki.fysik.dtu.dk/ase/index.html):

class UnifiedChemicalSystem

class UnifiedChemicalSystem(*system_block: str*)

A class representing a chemical system in the Amsterdam Modeling Suite.

to_ase_atoms() → Atoms

Converts a ChemicalSystem to an `ase.Atoms` instance.

Calling this method may throw an ImportError if the `ase` package can not be found in your Python environment.

Handles coordinates, atomic numbers, and lattice vectors (if present). The order of atoms is preserved. The systems total charge is saved as the `charge` entry in the `info` dictionary of the returned instance. All other molecular and atomic properties, bonds, regions and the atom selection is lost in the conversion.

classmethod from_ase_atoms(*ase_atoms: Atoms*) → *UnifiedChemicalSystem* (page 83)

Converts an `ase.Atoms` instance to a ChemicalSystem.

Calling this method may throw an ImportError if the `ase` package can not be found in your Python environment.

Handles coordinates, atomic numbers, and lattice vectors (if present). The order of atoms is preserved. If the `info` dictionary of the `Atoms` instance contains an entry `charge`, it is used as the ChemicalSystem's total charge. All other molecular or atomic properties, as well as the ASE calculator derived properties, such as forces or calculated charges are lost in the conversion.

Note that the used lattice vectors from the `ase.Atoms` object are those for which `get_pbc` is set to True. Those should already conform to the AMS convention of having the first lattice vector along the x-axis and the second vector in the xy-plane. A ChemicalSystemError will be thrown if that is not the case.

4.1.16 GUI and notebook integration

The ChemicalSystem class provides some convenience functions for integration with the AMS GUI programs, as well as Jupyter notebooks:

class UnifiedChemicalSystem

class UnifiedChemicalSystem(*system_block: str*)

A class representing a chemical system in the Amsterdam Modeling Suite.

gui() → int

Opens AMSinput to show the ChemicalSystem.

This will block the Python interpreter until the AMSinput process exits. Returns the exit code of the AMSinput process.

Note that atom selections are currently not shown in AMSinput.

plot(*figsize: Tuple[float, float] | None = (4, 4)*, *ax=None*, *keep_axis: bool = False*, ***kwargs*) → None

Shows a ChemicalSystem in a Jupyter notebook.

- `figsize` determines the size of the generated figure.
- `ax` can be used to place the figure into a Matplotlib subplot object.
- `keep_axis` determines the visibility of the axes of the plot.

The remaining keyword arguments of this method are forwarded to ASE's `plot_atoms` function, see the ASE documentation (https://wiki.fysik.dtu.dk/ase/ase/visualize/visualize.html#matplotlib) for details. A useful application of the keyword arguments is rotating the molecule in the plot:

```
mol = ChemicalSystem(...)
mol.plot(rotation="50x,40y,30z")
```

This method relies on ASE and matplotlib for the actual plotting and calling it may throw an ImportError if either of the two packages is can not be found in your Python environment.

4.2 Units

class Units

A utility class containing methods for units conversions.

Simple example usage:

```
from scm.libbase import Units

energies_in_hartree = [1,2,3]
energies_in_ev = Units.convert("Hartree", "eV", energies_in_hartree)

pressure_in_atm = 1.0
pressure_in_kpascal = pressure_in_atm * Units.get_factor('atm', 'kPascal')
```

classmethod convert (*from_unit: str*, *to_unit: str*, *value: int*) → float

classmethod convert (*from_unit: str*, *to_unit: str*, *value: float*) → float

classmethod convert (*from_unit: str*, *to_unit: str*, *value: number*) → number

classmethod convert (*from_unit: str*, *to_unit: str*, *value: List[int]*) → List[float]

classmethod convert (*from_unit: str*, *to_unit: str*, *value: List[float]*) → List[float]

classmethod convert (*from_unit: str*, *to_unit: str*, *value: List[number]*) → List[number]

classmethod convert (*from_unit: str*, *to_unit: str*, *value: List[Any]*) → List[Any]

classmethod convert (*from_unit: str*, *to_unit: str*, *value: Tuple[int, ...]*) → Tuple[float, ...]

classmethod convert (*from_unit: str*, *to_unit: str*, *value: Tuple[float, ...]*) → Tuple[float, ...]

classmethod convert (*from_unit: str*, *to_unit: str*, *value: Tuple[number, ...]*) → Tuple[number, ...]

classmethod convert (*from_unit: str*, *to_unit: str*, *value: Tuple[Any, ...]*) → Tuple[Any, ...]

classmethod convert (*from_unit: str*, *to_unit: str*, *value: ndarray[Any, dtype[ScalarType]]*) → ndarray[Any, dtype[ScalarType]]

Converts a value from one unit to another.

Parameters:

- from_unit (str): The unit of the input value. Must be a recognized unit.
- to_unit (str): The unit to convert the value to. Must be a recognized unit.
- value: The value to convert.

Returns:

- The value converted to the target unit. The type of the returned value will match the type of the input value.

Raises:

- UnitsError: If either *from_unit* or *to_unit* are not supported or if you are trying to perform a conversion between incompatible units.

Examples:

```
>>> from scm.libbase import Units

>>> Units.convert("m", "km", 1000)
1.0
>>> Units.convert("km", "m", [1,2,3])
[1000.0, 2000.0, 3000.0]
```

classmethod get_factor(*from_unit: str*, *to_unit: str*) → float

Returns the conversion factor between two units.

Parameters:

- from_unit (str): The unit you want to convert from. Must be a recognized unit.
- to_unit (str): The unit you want to convert to. Must be a recognized unit.

Returns:

- float: The conversion factor for converting *from_unit* to *to_unit*.

Raises:

- UnitsError: If either *from_unit* or *to_unit* is not supported or if you are trying to perform a conversion between incompatible units..

Examples:

```
>>> from scm.libbase import Units

>>> Units.get_factor("m", "km")
0.001
```

classmethod get_quantities() → List[str]

Returns all defined quantities. Quantities are things like 'LENGTH', 'ENERGY' or 'MASS'.

classmethod get_unit(*alt: str*) → str

Returns the unit for the given alternative representation.

Will throw on ambiguous result if there is no exact match.

classmethod get_units(*quantity: str*) → List[str]

Returns all units associated with a quantity. This will be in a nice unicode format.

classmethod units_summary_string() → str

Returns a string containing a nicely formatted table of all the units available.

class UnitsError

Error thrown for unrecognized units or illegal conversions.

4.2.1 List of available units

To see the list available units, you can use the *units_summary_string()* (page 71). Here is the output of `print(Units.units_summary_string())`:

```
======================
Quantity: ACCELERATION
======================

unit name                        label
-----------------------------------

angstrom/fs2                     Å/fs²


===============
Quantity: ANGLE
===============

unit name                        label
-----------------------------------

degree                           °
rad                              rad
grad                             grad
turn                             tr


=================
Quantity: CURRENT
=================

unit name                        label
-----------------------------------

ehartree/hbar                    Ha/ħ
ampere                           A
mampere                          mA
uampere                          µA


=================
Quantity: DENSITY
=================

unit name                        label
-----------------------------------

dalton/bohr3                     Da/Bohr³
kg/m3                            kg/m³
g/cm3                            g/cm³
kgl                              kg/L


================
Quantity: DIPOLE
================
```

(continues on next page)

(continued from previous page)

```
unit name                       label
-----------------------------------

ebohr                           Bohr
coulombmeter                    Cm
debye                           D


=======================
Quantity: ELECTRICFIELD
=======================

unit name                       label
-----------------------------------

hartree/ebohr                   Ha/Bohr
volt/angstrom                   V/Å
volt/meter                      V/m


===========================
Quantity: ELECTRICPOTENTIAL
===========================

unit name                       label
-----------------------------------

volt                            V


================
Quantity: ENERGY
================

unit name                       label
-----------------------------------

hartree                         Ha
rydberg                         Ry
joule                           J
kjoule                          kJ
ev                              eV
kcalmol                         kcal/mol
kjmol                           kJ/mol
cm-1                            cm^-1
mhz                             MHz
thz                             THz


================
Quantity: FORCES
================

unit name                       label
-----------------------------------

hartree/angstrom                Ha/Å
```

(continues on next page)

(continued from previous page)

```
hartree/meter                  Ha/m
hartree/bohr                   Ha/Bohr
rydberg/angstrom               Ry/Å
rydberg/meter                  Ry/m
rydberg/bohr                   Ry/Bohr
joule/angstrom                 J/Å
joule/m                        N
joule/bohr                     J/Bohr
kjoule/angstrom                kJ/Å
kjoule/meter                   kJ/m
kjoule/bohr                    kJ/Bohr
ev/angstrom                    eV/Å
ev/meter                       eV/m
ev/bohr                        eV/Bohr
kcalmol/angstrom               kcal/mol/Å
kcalmol/meter                  kcal/mol/m
kcalmol/bohr                   kcal/mol/Bohr
kjmol/angstrom                 kJ/mol/Å
kjmol/meter                    kJ/mol/m
kjmol/bohr                     kJ/mol/Bohr
cm-1/angstrom                  cm^-1/Å
cm-1/meter                     cm^-1/m
cm-1/bohr                      cm^-1/Bohr
mhz/angstrom                   MHz/Å
mhz/meter                      MHz/m
mhz/bohr                       MHz/Bohr
thz/angstrom                   THz/Å
thz/meter                      THz/m
thz/bohr                       THz/Bohr


================
Quantity: LENGTH
================

unit name                      label
-----------------------------------

bohr                           Bohr
pm                             pm
angstrom                       Å
nm                             nm
cm                             cm
dm                             dm
meter                          m


======================
Quantity: MAGNETICFIELD
======================

unit name                      label
-----------------------------------

gauss                          G
tesla                          T
aumagsi                        a.u. (mag SI)
```

(continues on next page)

(continued from previous page)

```
==============
Quantity: MASS
==============

unit name                       label
-----------------------------------

dalton                          Da
erestmass                       m_e
proton                          m□
g                               g
kg                              kg


=========================
Quantity: MOMENTOFINERTIA
=========================

unit name                       label
-----------------------------------

kgm2                            kgm²


==================
Quantity: PRESSURE
==================

unit name                       label
-----------------------------------

kpascal                         kPa
mpascal                         MPa
gpascal                         GPa
bar                             bar
mbar                            mbar
kbar                            kbar
atm                             atm
torr                            Torr
hartree/angstrom3               Ha/Å³
hartree/meter3                  Ha/m³
hartree/bohr3                   Ha/Bohr³
rydberg/angstrom3               Ry/Å³
rydberg/meter3                  Ry/m³
rydberg/bohr3                   Ry/Bohr³
joule/angstrom3                 J/Å³
pascal                          Pa
joule/bohr3                     J/Bohr³
kjoule/angstrom3                kJ/Å³
kjoule/meter3                   kJ/m³
kjoule/bohr3                    kJ/Bohr³
ev/angstrom3                    eV/Å³
ev/meter3                       eV/m³
ev/bohr3                        eV/Bohr³
kcalmol/angstrom3               kcal/mol/Å³
```

(continues on next page)

(continued from previous page)

```
kcalmol/meter3                kcal/mol/m³
kcalmol/bohr3                 kcal/mol/Bohr³
kjmol/angstrom3               kJ/mol/Å³
kjmol/meter3                  kJ/mol/m³
kjmol/bohr3                   kJ/mol/Bohr³
cm-1/angstrom3                cm⁻¹/Å³
cm-1/meter3                   cm⁻¹/m³
cm-1/bohr3                    cm⁻¹/Bohr³
mhz/angstrom3                 MHz/Å³
mhz/meter3                    MHz/m³
mhz/bohr3                     MHz/Bohr³
thz/angstrom3                 THz/Å³
thz/meter3                    THz/m³
thz/bohr3                     THz/Bohr³


==========================
Quantity: RECIPROCALLENGTH
==========================

unit name                     label
-----------------------------------

bohr1                         Bohr⁻¹
meter1                        m⁻¹
angstrom1                     Å⁻¹
nm1                           nm⁻¹


===================
Quantity: SHIELDING
===================

unit name                     label
-----------------------------------

ppm                           ppm


===================
Quantity: STIFFNESS
===================

unit name                     label
-----------------------------------

hartree/angstrom2             Ha/Å²
hartree/meter2                Ha/m²
hartree/bohr2                 Ha/Bohr²
rydberg/angstrom2             Ry/Å²
rydberg/meter2                Ry/m²
rydberg/bohr2                 Ry/Bohr²
joule/angstrom2               J/Å²
joule/m2                      J/m²
joule/bohr2                   J/Bohr²
kjoule/angstrom2              kJ/Å²
kjoule/meter2                 kJ/m²
```

(continues on next page)

(continued from previous page)

```
kjoule/bohr2                    kJ/Bohr²
ev/angstrom2                    eV/Å²
ev/meter2                       eV/m²
ev/bohr2                        eV/Bohr²
kcalmol/angstrom2               kcal/mol/Å²
kcalmol/meter2                  kcal/mol/m²
kcalmol/bohr2                   kcal/mol/Bohr²
kjmol/angstrom2                 kJ/mol/Å²
kjmol/meter2                    kJ/mol/m²
kjmol/bohr2                     kJ/mol/Bohr²
cm-1/angstrom2                  cm⁻¹/Å²
cm-1/meter2                     cm⁻¹/m²
cm-1/bohr2                      cm⁻¹/Bohr²
mhz/angstrom2                   MHz/Å²
mhz/meter2                      MHz/m²
mhz/bohr2                       MHz/Bohr²
thz/angstrom2                   THz/Å²
thz/meter2                      THz/m²
thz/bohr2                       THz/Bohr²


=====================
Quantity: TEMPERATURE
=====================

unit name                       label
-----------------------------------

kelvin                          K
celsius                         °C
fahrenheit                      °F


==============
Quantity: TIME
==============

unit name                       label
-----------------------------------

s                               s
ms                              ms
us                              μs
ns                              ns
ps                              ps
fs                              fs
hartree_au_time                 ℏ/Ha
rydberg_au_time                 ℏ/Ry
hr                              hr
min                             min


==================
Quantity: VELOCITY
==================

unit name                       label
```

(continues on next page)

(continued from previous page)

```
-----------------------------------

bohr/fs                          Bohr/fs


================
Quantity: VOLUME
================

unit name                        label
-----------------------------------

bohr3                            Bohr³
angstrom3                        Å³
m3                               m³
```

4.3 Input File

class InputFile(*program: str*, *text_input: str = ''*, *string_leafs: bool = False*, *lowercase: bool = False*)

Class representing a key and block based text input file as used in the Amsterdam Modeling Suite.

class EntryCategory(*value*)

Enum representing the category of an entry in an InputFile.

Values:

- Block: the entry is a block containing keys and other blocks
- Key: the entry is a key with a value

class EntryType(*value*)

Enum representing the type of an entry in an InputFile.

Values:

- FixedBlock: the entry is a block containing defined keys and other blocks
- FreeBlock: the entry is a free block that is read as a list of strings representing the lines
- EngineBlock: the entry is a free block that ends with “EndEngine”
- InputBlock: the entry is a free block that ends with “EndInput”
- StringKey: the entry is a key of type string
- MultipleChoiceKey: the entry is a multiple-choice key
- LogicalKey: the entry is a key of type logical
- IntegerKey: the entry is a key of type integer
- IntegerListKey: the entry is a key of type integer_list
- RealKey: the entry is a key of type float
- RealListKey: the entry is a key of type float_list

get(*query: str*) → int | float | bool | str | List[int] | List[float] | List[str]

Returns the value of a key or free block. The return type depends on the type of the accessed key.

get_category (*query: str*) → *EntryCategory* (page 78)
Returns the category of an entry as defined in the JSON definition for the program.

get_entries_in_order (*query: str*) → List[str]
Returns the entries of an ordered fixed block in the order they occurred in the input.

get_freeblock (*query: str*) → List[str]
Returns the contents of a free block as a list of lines.

get_header (*query: str*) → str
Returns the header of a block as a string. If a block does not have a header, an empty string is returned.

get_int (*query: str*) → int
Returns the value of an integer key.

get_int_list (*query: str*) → List[int]
Returns the value of an integer_list key.

get_json () → str
Returns the JSON representation of the input file as a string.

get_logical (*query: str*) → bool
Returns the value of a logical key.

get_real (*query: str*) → float
Returns the value of a float key.

get_real_list (*query: str*) → List[float]
Returns the value of a float_list key.

get_string (*query: str*) → str
Returns the value of a string or multiple choice key.

get_system_blocks_as_mols () → Dict[str, Molecule]
Returns a dictionary of mapping the System blocks in the input to plams.Molecule instances.

The keys in the dictionary are the names of the systems from the headers of the blocks, e.g.:

```
System initial
    ...
End
```

would be returned as dict["initial"]. The main system without a name in the header will have the empty string as key.

get_type (*query: str*) → *EntryType* (page 78)
Returns the type of an entry as defined in the JSON definition for the program.

is_defined (*query: str*) → bool
Checks if an entry is defined in the JSON definition for the program.

is_ordered (*query: str*) → bool
Checks if a fixed block is declared order sensitive in the JSON definition for the program.

is_present (*query: str*) → bool
Checks if an entry is present in the input.

is_unique (*query: str*) → bool
Check whether an entry is declared unique or recurring in the keywords definition file.

number_of_entries (*query: str*) → int

Returns the number of occurrances of a non-unique entry.

read (*text_input: str*, *string_leafs: bool = False*, *lowercase: bool = False*) → None

Read more text input for the program. This is faster than constructing a new InputFile from scratch, as it will not reread the JSON definitions.

property program: str

The name of the program that this instance is reading input for.

class InputError

Error thrown for issues with the input reading, e.g. reading non-existing keywords.

4.4 KFFile

See also:

The *AKFReader* (page 101) Python class provides a convenient alternative for reading data from KF files.

class KFFile (*filename: str*, *omode:* OpenMode *(page 80) = OpenMode.Any*)

A class representing a file in KF format. Allows inspecting, reading and writing KF files.

class OpenMode (*value*)

Enum representing the mode in which a file should be opened.

Values:

- Any: Open an existing file or create a new file.
- New: Creates a new file. If the file already exists, a KFError exception may be thrown.
- Old: Open an existing file. If the file does not exist, a KFError exception may be thrown.

class StringSep (*value*)

Enum representing the separator used to delimit elements in arrays of strings:

Values:

- NullChar: The strings are separated with the C-style \0 character.
- NewLine: The strings are separated with the \n newline character.
- LCHARS: When writing each string is padded or truncated to a width of LCHARS=160 characters. The total length of the variable is thus a multiple of 160. When reading, the variable is split into sections of 160 characters again, representing the individual elements of the array.

class VarType (*value*)

Enum representing type of a variable on a KF file.

Values:

- Empty: indicates an empty index slot
- Int: a variable of integer type
- Real: a variable of real type
- Char: a variable of string type
- Logical: a variable of logical type

close() → None

Closes the file. No further methods should be called on a file that has been closed.

close_var() → None

Closes a variable.

copy_section(*from_kf:* KFFile *(page 80)*, *to_kf:* KFFile *(page 80)*, *section: str*) → None

copy_section(*from_kf:* KFFile *(page 80)*, *to_kf:* KFFile *(page 80)*, *from_section: str*, *to_section: str*) → None

Helper for @overload to raise when called.

create_section(*section: str*) → None

Creates a new section on the KF file.

create_var(*section: str*, *variable: str*, *n: int*, *type:* VarType *(page 80)*) → None

Creates a new variable on the file of a given length and type.

classmethod delete_file(*filename: str*) → None

Deletes a KF file from disk after checking that no other KFFile instances reference that file.

delete_section(*section: str*) → None

Deletes a section and all of the contained variables from the file.

delete_var(*section: str*, *variable: str*) → None

Deletes a variable from the KF file.

get_skeleton() → Dict[str, Set[str]]

Return a dictionary reflecting the structure of this KF file.

Each key in that dictionary corresponds to a section name of the KF file with the value being a set of variable names.

classmethod is_kffile(*filename: str*) → bool

Checks if a file has the KF format, aka it is a properly formatted KF file.

num_sections() → int

Returns the number of sections existing on the file.

num_variables(*section: str*) → int

Returns the number of variables in a section.

open_var(*section: str*, *variable: str*) → None

Opens a variable for reading and writing. Mostly useful if you incrementally want to read/write data from/to it.

read(*section: str = ''*, *variable: str = ''*, *return_as_list: bool = False*, *string_separator:* StringSep *(page 80)* | *None = None*) → bool | int | float | str | List[bool] | List[int] | List[float] | List[str]

Extract and return data for a variable located in a section.

By default, for single-value numerical or boolean variables returned value is a single number or bool. For longer variables this method returns a list of values. This behavior can be changed by setting return_as_list parameter to `True`. In that case the returned value is always a list of numbers (possibly of length 1).

For string variables this works a bit differently: the return_as_list argument is ignored. Instead an optional KFFile.StringSep can be given as a separator. If any separator is give, the return value will always be a list of strings (possibly of length 1 if there is nothing to be separated). If no separator is given, the return value is always a single string.

This method exists primarily for backwards compatibility with the plams.KFFile. For new code we recommend using the strongly typed read_*() methods instead, e.g. read_int().

read_int (*section: str*, *variable: str*) → int

Reads a single integer from a specific section and variable.

read_ints (*section: str*, *variable: str*) → List[int]

Reads a list of integers from a specific section and variable.

read_ints_np (*section: str*, *variable: str*) → ndarray[Any, dtype[int64]]

Reads integers from a specific section and variable into a numpy array.

read_logical (*section: str*, *variable: str*) → bool

Reads a single logical value from a specific section and variable.

read_logicals (*section: str*, *variable: str*) → List[bool]

Reads a list of logical values from a specific section and variable.

read_logicals_np (*section: str*, *variable: str*) → ndarray[Any, dtype[bool_]]

Reads logical values from a specific section and variable into a numpy array.

read_real (*section: str*, *variable: str*) → float

Reads a single real number from a specific section and variable.

read_reals (*section: str*, *variable: str*) → List[float]

Reads a list of real numbers from a specific section and variable.

read_reals_np (*section: str*, *variable: str*) → ndarray[Any, dtype[float64]]

Reads real numbers from a specific section and variable into a numpy array.

read_section (*section: str*) → Dict[str, bool | int | float | str | List[bool] | List[int] | List[float] | List[str]]

Return a dictionary with all variables from a given section.

Note: Some sections can contain very large amount of data.
Turning them into dictionaries can cause memory shortage or performance issues. Use this method carefully …

read_string (*section: str*, *variable: str*) → str

Reads a single string from a specific section and variable.

read_strings (*section: str*, *variable: str*, *sep:* StringSep *(page 80)* = *StringSep.LCHARS*) → List[str]

Reads a list of strings from a specific section and variable.

rewind_var () → None

Rewinds a variable. Incremental reading and writing will then start from the beginning of the variable again.

save () → None

Write all internal buffers to disk (or at least the OS cache). It's basically a flush …

section_exists (*section: str*) → bool

Check if a section exists on the file.

sections () → List[str]

Returns a list of all sections on the file.

skip_var (*n: int*) → None

Skip forward over a number of elements in the variable.

var_exists (*section: str*, *variable: str*) → bool

Check if a variable exists on the file.

var_length (*section: str*, *variable: str*) → int

Returns the allocated length of variable on file. This is the maximum number of elements that can be written to it without reallocation.

var_type (*section: str*, *variable: str*) → *VarType* (page 80)

Returns the type of the elements of a variable.

var_used (*section: str*, *variable: str*) → int

Returns the number of elements written to a variable.

variables (*section: str*) → List[str]

Returns a list of all variables in a section.

write (*value: bool*) → None

write (*value: int*) → None

write (*value: float*) → None

write (*value: str*) → None

write (*value: List[bool]*) → None

write (*value: List[int]*) → None

write (*value: List[float]*) → None

write (*value: List[str]*) → None

write (*section: str*, *variable: str*, *value: bool*) → None

write (*section: str*, *variable: str*, *value: int*) → None

write (*section: str*, *variable: str*, *value: float*) → None

write (*section: str*, *variable: str*, *value: str*) → None

write (*section: str*, *variable: str*, *value: List[bool]*) → None

write (*section: str*, *variable: str*, *value: List[int]*) → None

write (*section: str*, *variable: str*, *value: List[float]*) → None

write (*section: str*, *variable: str*, *value: List[str]*, *sep:* StringSep *(page 80)* = *StringSep.LCHARS*) → None

Helper for @overload to raise when called.

property filename: str

The filename of the file managed by this KFFile instance.

property path: str

The absolute path to the file managed by this KFFile instance.

class KFError

Error thrown for all KF issues, e.g. reading of non-existing variables or reading as wrong datatypes.

4.5 LibBase API index

class UnifiedChemicalSystem

A class representing a chemical system in the Amsterdam Modeling Suite

__init__ (**args*, ***kwargs*)

Overloaded function.

1. __init__(self: scm_libbase_internal.UnifiedChemicalSystem) -> None

Creates a new, empty ChemicalSystem.

2. __init__(self: scm_libbase_internal.UnifiedChemicalSystem, system_block: str) -> None

Creates a new ChemicalSystem from a System block string.

__len__ (*self:* UnifiedChemicalSystem *(page 83)*) → int

__str__ (*self:* UnifiedChemicalSystem *(page 83)*) → str

Formats a ChemicalSystem into an AMS System block.

add_all_atoms_to_region (*self:* UnifiedChemicalSystem *(page 83)*, *region: str*) → None

Adds all of the system's atoms to a region.

add_atom (**args*, ***kwargs*)

Overloaded function.

1. add_atom(self: scm_libbase_internal.UnifiedChemicalSystem, atom: scm_libbase_internal.UnifiedAtom) -> None

Adds a copy of an Atom to a ChemicalSystem.

2. add_atom(self: scm_libbase_internal.UnifiedChemicalSystem, atom: scm_libbase_internal.UnifiedAtom, coords: numpy.ndarray[numpy.float64[3, 1]], unit: str = 'angstrom') -> None

Adds a copy of an Atom to a ChemicalSystem.

3. add_atom(self: scm_libbase_internal.UnifiedChemicalSystem, Z: int, coords: numpy.ndarray[numpy.float64[3, 1]], unit: str = 'angstrom') -> None

Adds a new Atom of a given atomic number to the ChemicalSystem.

4. add_atom(self: scm_libbase_internal.UnifiedChemicalSystem, element: scm_libbase_internal.UnifiedElement, coords: numpy.ndarray[numpy.float64[3, 1]], unit: str = 'angstrom') -> None

Adds a new Atom of a given Element to the ChemicalSystem.

5. add_atom(self: scm_libbase_internal.UnifiedChemicalSystem, symbol: str, coords: numpy.ndarray[numpy.float64[3, 1]], unit: str = 'angstrom') -> None

Adds a new Atom to the ChemicalSystem given its symbol.

add_atom_to_region (**args*, ***kwargs*)

Overloaded function.

1. add_atom_to_region(self: scm_libbase_internal.UnifiedChemicalSystem, atom: int, region: str) -> None

Adds an atom to a region.

2. add_atom_to_region(self: scm_libbase_internal.UnifiedChemicalSystem, atom: scm_libbase_internal.UnifiedAtom, region: str) -> None

Adds an atom to a region.

add_atoms_to_region (*self:* UnifiedChemicalSystem *(page 83)*, *atom_indices: numpy.ndarray[numpy.int32[m, 1]]*, *region: str*) → None

Adds multiple atoms to a region, given their atom indices.

add_hydrogen_atoms (**args*, ***kwargs*)

Overloaded function.

1. add_hydrogen_atoms(self: scm_libbase_internal.UnifiedChemicalSystem, atom_indices: List[int]) -> None

Add missing hydrogen atoms to organic compounds.

2. add_hydrogen_atoms(self: scm_libbase_internal.UnifiedChemicalSystem) -> None

Add missing hydrogen atoms to organic compounds.

add_other (*self:* UnifiedChemicalSystem *(page 83)*, *other:* UnifiedChemicalSystem *(page 83)*) → None

Merges another ChemicalSystem into this one.

align_to (*self:* UnifiedChemicalSystem *(page 83)*, *other:* UnifiedChemicalSystem *(page 83)*) → None

Translate and rotate the system to maximally align it with 'other'.

static all_from_input (*input_file: scm_libbase_internal.InputFile*) → Dict[str, *UnifiedChemicalSystem* (page 83)]

Constructs and returns ChemicalSystems for all System (and LoadSystem) blocks in an InputFile.

apply_strain (*self:* UnifiedChemicalSystem *(page 83)*, *strain_matrix: numpy.ndarray[numpy.float64[m, n]]*) → None

Apply a strain deformation to a periodic system.

apply_strain_voigt (*self:* UnifiedChemicalSystem *(page 83)*, *strain_voigt: List[float]*) → None

Apply a strain deformation to a periodic system.

atom_index (*self:* UnifiedChemicalSystem *(page 83)*, *atom: scm_libbase_internal.UnifiedAtom*) → int

Given an Atom instance, returns its index in ChemicalSystem.atoms.

atom_is_in_ring (**args*, ***kwargs*)

Overloaded function.

1. atom_is_in_ring(self: scm_libbase_internal.UnifiedChemicalSystem, atom: int) -> bool

Checks if an atom is part of any ring.

2. atom_is_in_ring(self: scm_libbase_internal.UnifiedChemicalSystem, atom: scm_libbase_internal.UnifiedAtom) -> bool

Checks if an atom is part of any ring.

atomic_properties_enabled (*self:* UnifiedChemicalSystem *(page 83)*, *group_prefix: str*) → bool

Checks if a group of atomic properties is enabled or not.

bond_cuts_molecule (**args*, ***kwargs*)

Overloaded function.

1. bond_cuts_molecule(self: scm_libbase_internal.UnifiedChemicalSystem, from_atom: int, to_atom: int) -> bool

Checks if removing the bonds between two atoms would cut the graph into two disjoint subgraphs.

2. bond_cuts_molecule(self: scm_libbase_internal.UnifiedChemicalSystem, from_atom: scm_libbase_internal.UnifiedAtom, to_atom: scm_libbase_internal.UnifiedAtom) -> bool

Checks if removing the bonds between two atoms would cut the graph into two disjoint subgraphs.

bounding_box_volume (*self:* UnifiedChemicalSystem *(page 83)*, *unit: str = 'angstrom3'*) → float

Volume of the bounding box.

center_of_mass (*self:* UnifiedChemicalSystem *(page 83)*, *unit: str = 'angstrom'*) → numpy.ndarray[numpy.float64[3, 1]]

Position of the center of mass.

check_molecule_symmetry (*self:* UnifiedChemicalSystem *(page 83)*, *label: str*, *tolerance: float = 1e-07*) → bool

Checks if a molecule has a particular symmetry given by a Schoenflies symbol.

contains_atom (*self:* UnifiedChemicalSystem *(page 83)*, *atom: scm_libbase_internal.UnifiedAtom*) → bool

Checks if an Atom instance is part of a ChemicalSystem.

copy (*self:* UnifiedChemicalSystem *(page 83)*) → *UnifiedChemicalSystem* (page 83)

Creates a deep copy of the ChemicalSystem.

density (*self:* UnifiedChemicalSystem *(page 83)*, *unit: str = 'dalton/angstrom3'*) → float

Returns the density of the system in the specified unit. Only valid for 3D periodic systems.

deselect_all (*self:* UnifiedChemicalSystem *(page 83)*) → None

Deselects all atoms, or in other words: clears the current selection.

deselect_atom (**args*, ***kwargs*)

Overloaded function.

1. deselect_atom(self: scm_libbase_internal.UnifiedChemicalSystem, arg0: int) -> None

Selects an atom given its index.

2. deselect_atom(self: scm_libbase_internal.UnifiedChemicalSystem, arg0: scm_libbase_internal.UnifiedAtom) -> None

Selects an instance of an atom.

deselect_atoms (*self:* UnifiedChemicalSystem *(page 83)*, *atom_indices: numpy.ndarray[numpy.int32[m, 1]]*) → None

Deselects multiple atoms at once, given their atom indices.

deselect_atoms_if (*self:* UnifiedChemicalSystem *(page 83)*, *pred: Callable[[scm_libbase_internal.UnifiedAtom], bool]*) → None

Deselects atoms based on a predicate function.

determine_species (*self:* UnifiedChemicalSystem *(page 83)*, *comp: Callable[[scm_libbase_internal.UnifiedAtom, scm_libbase_internal.UnifiedAtom], bool]*) → Tuple[int, List[int], List[int]]

Determines the present atomic species, based on a user defined comparator function.

disable_atomic_properties (*self:* UnifiedChemicalSystem *(page 83)*, *group_prefix: str*) → None

Disables the use of a group of atomic properties. Any set properties within the group will be discarded.

do_regions_intersect (*self:* UnifiedChemicalSystem *(page 83)*, *regionA: str*, *regionB: str*) → bool

Checks if two regions or region expressions intersect, i.e. have at least one atom in common.

enable_atomic_properties (*self:* UnifiedChemicalSystem *(page 83)*, *group_prefix: str*) → None

Enables the use of a group of atomic properties.

extract_atoms (*self:* UnifiedChemicalSystem *(page 83)*, *atom_indices: List[int]*) → *UnifiedChemicalSystem* (page 83)

Returns a new system build from a subset of atoms.

formula (*self:* UnifiedChemicalSystem *(page 83)*) → str

Chemical formula in Hill notation, e.g. H2O or CH4.

static from_in (*filename: str*, *name: str = ''*) → *UnifiedChemicalSystem* (page 83)

Constructs and returns a new ChemicalSystem from a (possibly named) System block in an AMS input file.

static from_input (*input_file: scm_libbase_internal.InputFile*, *prefix: str*) → *UnifiedChemicalSystem* (page 83)

Constructs and returns a new ChemicalSystem from an InputFile instance, given a prefix, e.g. 'System[1]%'.

static from_kf (**args*, ***kwargs*)

Overloaded function.

1. from_kf(filename: str, section: str = 'Molecule') -> scm_libbase_internal.UnifiedChemicalSystem

Constructs and returns a new ChemicalSystem from a section on a KF file.

2. from_kf(kf: scm_libbase_internal.KFFile, section: str = 'Molecule') -> scm_libbase_internal.UnifiedChemicalSystem

Constructs and returns a new ChemicalSystem from a section on a KF file.

static from_xyz (*filename: str*) → *UnifiedChemicalSystem* (page 83)

Constructs and returns a new ChemicalSystem from an extended XYZ file.

geometric_center (*self:* UnifiedChemicalSystem *(page 83)*, *unit: str = 'angstrom'*) → numpy.ndarray[numpy.float64[3, 1]]

Position of the geometric center.

get_atoms_in_region (*self:* UnifiedChemicalSystem *(page 83)*, *region: str*) → numpy.ndarray[numpy.int32[m, 1]]

Returns a sorted array of atom indices of all atoms in a region or region expression.

get_atoms_outside_region (*self:* UnifiedChemicalSystem *(page 83)*, *region: str*) → numpy.ndarray[numpy.int32[m, 1]]

Returns a sorted array of atom indices of all atoms outside of a region or region expression.

get_distance (**args*, ***kwargs*)

Overloaded function.

1. get_distance(self: scm_libbase_internal.UnifiedChemicalSystem, a: int, b: int, unit: str = 'angstrom') -> float

Measures the distance between two atoms.

2. get_distance(self: scm_libbase_internal.UnifiedChemicalSystem, a: scm_libbase_internal.UnifiedAtom, b: scm_libbase_internal.UnifiedAtom, unit: str = 'angstrom') -> float

Measures the distance between two atoms.

get_fractional_coordinate (*self:* UnifiedChemicalSystem *(page 83)*, *atom: scm_libbase_internal.UnifiedAtom*, *unit: str = 'angstrom'*) → numpy.ndarray[numpy.float64[3, 1]]

Returns the fractional coordinate of a single atom with respect to the cell. In non-periodic directions the plain coordinates are returned in the specified unit.

get_fractional_coordinates (*self:* UnifiedChemicalSystem *(page 83)*, *unit: str = 'angstrom'*) → numpy.ndarray[numpy.float64[m, 3]]

Returns the fractional coordinates of all atoms with respect to the cell. In non-periodic directions the plain coordinates are returned in the specified unit.

static get_operands_in_region_expression (*region_expression: str*) → List[str]

Returns a sorted list of all unique operands used in a region expression.

get_regions_of_atom (**args*, ***kwargs*)

Overloaded function.

1. get_regions_of_atom(self: scm_libbase_internal.UnifiedChemicalSystem, atom: int) -> List[str]

Returns an alphabetically sorted list of the names of all regions an atom is part of.

2. get_regions_of_atom(self: scm_libbase_internal.UnifiedChemicalSystem, atom: scm_libbase_internal.UnifiedAtom) -> List[str]

Returns an alphabetically sorted list of the names of all regions an atom is part of.

get_selected_atoms (*self:* UnifiedChemicalSystem *(page 83)*) → numpy.ndarray[numpy.int32[m, 1]]

Returns an array of indices of all selected atoms.

get_species (*self:* UnifiedChemicalSystem *(page 83)*, *comp: Callable[[scm_libbase_internal.UnifiedAtom, scm_libbase_internal.UnifiedAtom], bool]*) → List[List[int]]

Determines the present atomic species, based on a user defined comparator function.

guess_bonds (*self:* UnifiedChemicalSystem *(page 83)*) → None

Guesses bonds based on the atomic elements and the geometry. Keeps existing bonds.

has_bonds (*self:* UnifiedChemicalSystem *(page 83)*) → bool

Checks if the system has any bonding information.

has_ghost_atoms (*self:* UnifiedChemicalSystem *(page 83)*) → bool

Checks whether the ChemicalSystem contains any Ghost atoms.

has_lattice (*self:* UnifiedChemicalSystem *(page 83)*) → bool

Checks if the system has a lattice.

has_region (*self:* UnifiedChemicalSystem *(page 83)*, *region: str*) → bool

Checks if a region exists in the system, given its region name.

has_same_atoms (**args*, ***kwargs*)

Overloaded function.

1. has_same_atoms(self: scm_libbase_internal.UnifiedChemicalSystem, other: scm_libbase_internal.UnifiedChemicalSystem, properties: List[str] = []) -> bool

Checks if two systems have identical atoms in the same order.

2. has_same_atoms(self: scm_libbase_internal.UnifiedChemicalSystem, other: scm_libbase_internal.UnifiedChemicalSystem, comp: Callable[[scm_libbase_internal.UnifiedAtom, scm_libbase_internal.UnifiedAtom], bool]) -> bool

Checks if two systems have identical atoms in the same order given a user defined comparator.

has_same_coords (*self:* UnifiedChemicalSystem *(page 83)*, *other:* UnifiedChemicalSystem *(page 83)*, *tol: float = 0.001*, *unit: str = 'angstrom'*) → bool

Checks if the atomic coordinates of two systems are within a threshold of each other.

has_same_geometry (*self:* UnifiedChemicalSystem *(page 83)*, *other:* UnifiedChemicalSystem *(page 83)*, *tol: float = 0.001*, *unit: str = 'angstrom'*) → bool

Checks if the atomic coordinates and lattice vectors of two systems are within a threshold of each other.

has_same_regions (*self:* UnifiedChemicalSystem *(page 83)*, *other:* UnifiedChemicalSystem *(page 83)*) → bool

Checks if two systems have identical regions, meaning region names and included atom indices.

has_same_selection (*self:* UnifiedChemicalSystem *(page 83)*, *other:* UnifiedChemicalSystem *(page 83)*, *consider_selection_order: bool = False*) → bool

Checks if two systems have the same atom selection.

inertia_tensor (*self:* UnifiedChemicalSystem *(page 83)*) → numpy.ndarray[numpy.float64[3, 3]]

Returns the system's inertia tensor as a 3x3 matrix (units amu*angstrom^2).

invert_selection (*self:* UnifiedChemicalSystem *(page 83)*) → None

Inverts the set of selected atoms.

is_atom_in_region (**args*, ***kwargs*)

Overloaded function.

1. is_atom_in_region(self: scm_libbase_internal.UnifiedChemicalSystem, atom: int, region: str) -> bool

Checks if an atom is in a region or region expression.

2. is_atom_in_region(self: scm_libbase_internal.UnifiedChemicalSystem, atom: scm_libbase_internal.UnifiedAtom, region: str) -> bool

Checks if an atom is in a region or region expression.

is_atom_outside_region (**args*, ***kwargs*)

Overloaded function.

1. is_atom_outside_region(self: scm_libbase_internal.UnifiedChemicalSystem, atom: int, region: str) -> bool

Checks if an atom is outside of a region or region expression.

2. is_atom_outside_region(self: scm_libbase_internal.UnifiedChemicalSystem, atom: scm_libbase_internal.UnifiedAtom, region: str) -> bool

Checks if an atom is outside of a region or region expression.

is_atom_selected (**args*, ***kwargs*)

Overloaded function.

1. is_atom_selected(self: scm_libbase_internal.UnifiedChemicalSystem, arg0: int) -> bool

Checks if an atom is currently selected, given its index.

2. is_atom_selected(self: scm_libbase_internal.UnifiedChemicalSystem, arg0: scm_libbase_internal.UnifiedAtom) -> None

Checks if an instance of an atom is currently selected.

is_linear (*self:* UnifiedChemicalSystem *(page 83)*) → bool

Checks is a ChemicalSystem is linear.

static is_valid_region_name (*name: str*) → bool

Checks if a string is a valid region name.

make_conventional_cell (*self:* UnifiedChemicalSystem *(page 83)*, *precision: float = 0.1*) → *UnifiedChemicalSystem* (page 83)

Create a new system by converting a 2D or 3D cell to its conventional representation.

make_primitive_cell (*self:* UnifiedChemicalSystem *(page 83)*, *precision: float = 0.1*) → *UnifiedChemicalSystem* (page 83)

Create a new system by converting a 2D or 3D cell to its primitive representation.

make_selection_cappable (*self:* UnifiedChemicalSystem *(page 83)*) → None

Extends the current selection but does not cross single bonds, unless they are to hydrogen atoms.

make_slab_layers (*self:* UnifiedChemicalSystem *(page 83)*, *ref_atom: int*, *num_layers: int*, *miller: numpy.ndarray[numpy.int32[3, 1]]*, *translate: float = 0.0*, *unit: str = 'angstrom'*) → *UnifiedChemicalSystem* (page 83)

Create a new system which is a 2D slab, by slicing a 3D system, specifying the number of layers in the resulting slab.

make_slab_thickness (*self:* UnifiedChemicalSystem *(page 83)*, *ref_atom: int*, *top: float*, *bottom: float*, *miller: numpy.ndarray[numpy.int32[3, 1]]*, *translate: float = 0.0*, *unit: str = 'angstrom'*) → *UnifiedChemicalSystem* (page 83)

Create a new system which is a 2D slab, by slicing a 3D system, specifying the upper and lower bound of the resulting slab.

make_supercell (*self:* UnifiedChemicalSystem *(page 83)*, *supercell: numpy.ndarray[numpy.int32[m, 1]]*) → *UnifiedChemicalSystem* (page 83)

Create a supercell by scaling the lattice vectors.

make_supercell_trafo (*self:* UnifiedChemicalSystem *(page 83)*, *supercell: numpy.ndarray[numpy.int32[m, n]]*) → *UnifiedChemicalSystem* (page 83)

Create a supercell by creating the matrix product of the lattice vectors and the supercell matrix.

map_atoms (**args*, ***kwargs*)

Overloaded function.

1. map_atoms(self: scm_libbase_internal.UnifiedChemicalSystem, start_range: float) -> Tuple[bool, numpy.ndarray[numpy.int32[m, 3]]]

Maps all atoms into a unit cell from [start_range:start_range+1] in fractional coordinates.

2. map_atoms(self: scm_libbase_internal.UnifiedChemicalSystem, start_range: numpy.ndarray[numpy.float64[m, 1]]) -> Tuple[bool, numpy.ndarray[numpy.int32[m, 3]]]

Maps all atoms into a unit cell from [start_range:start_range+1] in fractional coordinates.

map_atoms_around_atom (**args*, ***kwargs*)

Overloaded function.

1. map_atoms_around_atom(self: scm_libbase_internal.UnifiedChemicalSystem, atom: scm_libbase_internal.UnifiedAtom) -> Tuple[bool, numpy.ndarray[numpy.int32[m, 3]]]

Map all atoms around the chosen atom that will be in the center of the new unit cell.

2. map_atoms_around_atom(self: scm_libbase_internal.UnifiedChemicalSystem, atom: int) -> Tuple[bool, numpy.ndarray[numpy.int32[m, 3]]]

Map all atoms around the chosen atom that will be in the center of the new unit cell.

map_atoms_continuous (*self:* UnifiedChemicalSystem *(page 83)*) → bool

Map all atoms that are bonded across the periodic boundary back.

molecule_indices (*self:* UnifiedChemicalSystem *(page 83)*) → numpy.ndarray[numpy.int32[m, 1]]

Returns a `num_atoms` sized array mapping the atoms to connected molecules of the system.

molgraph_dijkstra (*self:* UnifiedChemicalSystem *(page 83)*, *from_atidx: int*, *dist_func: Callable[[int, int, scm_libbase_internal.UnifiedBond], float]*, *to_atidx: int = -1*) → Tuple[List[float], List[int], List[bool]]

General method implementing the Dijkstra algorithm on the molecular graph.

moments_of_inertia (*self:* UnifiedChemicalSystem *(page 83)*) → Tuple[numpy.ndarray[numpy.float64[3, 1]], numpy.ndarray[numpy.float64[3, 3]]]

Calculates the system's moments of inertia and the corresponding principal axes (units amu*angstrom^2).

moving_atoms_for_angle_change (*self: scm_libbase_internal.UnifiedChemicalSystem*, *a: int*, *b: int*, *c: int*, *policy: scm_libbase_internal.UnifiedChemicalSystem.InternalCoordinateManipulationPolicy = <InternalCoordinateManipulationPolicy.SecondPartMoves: 1>*) → numpy.ndarray[numpy.int32[m, 1]]

Determines which atoms will move when changing the angle between three atoms.

moving_atoms_for_dihedral_change (*self: scm_libbase_internal.UnifiedChemicalSystem*, *a: int*, *b: int*, *c: int*, *d: int*, *policy: scm_libbase_internal.UnifiedChemicalSystem.InternalCoordinateManipulationPoli = <InternalCoordinateManipulationPolicy.SecondPartMoves: 1>*) → numpy.ndarray[numpy.int32[m, 1]]

Determines which atoms will move when changing the dihedral between four atoms.

moving_atoms_for_distance_change (*self: scm_libbase_internal.UnifiedChemicalSystem*, *a: int*, *b: int*, *policy: scm_libbase_internal.UnifiedChemicalSystem.InternalCoordinateManipulationPoli = <InternalCoordinateManipulationPolicy.LighterPartMoves: 2>*) → numpy.ndarray[numpy.int32[m, 1]]

Determines which atoms will move when changing the distance between two atoms.

num_atoms_in_region (*self:* UnifiedChemicalSystem *(page 83)*, *region: str*) → int

Returns the number of atoms in a region or region expression.

num_atoms_outside_region (*self:* UnifiedChemicalSystem *(page 83)*, *region: str*) → int

Returns the number of atoms outside of a region or region expression.

num_molecules (*self:* UnifiedChemicalSystem *(page 83)*) → int

Counts the number of connected molecules that are part of this system.

num_selected_atoms (*self:* UnifiedChemicalSystem *(page 83)*) → int

The number of currently selected atoms.

perturb_coordinates (*self:* UnifiedChemicalSystem *(page 83)*, *noise_level: float*, *unit: str = 'angstrom'*) → None

Perturb the atomic coordinates by adding random numbers between to each Cartesian component.

perturb_lattice (*self:* UnifiedChemicalSystem *(page 83)*, *noise_level: float*) → None

Perturb the lattice vectors by applying a random strain.

remove_all_regions (*self:* UnifiedChemicalSystem *(page 83)*) → None

Removes all regions. All atoms will keep existing, but will not longer be part of any region.

remove_atom (*self:* UnifiedChemicalSystem *(page 83)*, *atom_index: int*) → None

Removes a single atom from the system, given its index.

remove_atom_from_all_regions (**args*, ***kwargs*)

Overloaded function.

1. remove_atom_from_all_regions(self: scm_libbase_internal.UnifiedChemicalSystem, atom: int) -> None

Removes an atom from all regions.

2. remove_atom_from_all_regions(self: scm_libbase_internal.UnifiedChemicalSystem, atom: scm_libbase_internal.UnifiedAtom) -> None

Removes an atom from all regions.

remove_atom_from_region (**args*, ***kwargs*)

Overloaded function.

1. remove_atom_from_region(self: scm_libbase_internal.UnifiedChemicalSystem, atom: int, region: str) -> None

Removes an atom from a region.

2. remove_atom_from_region(self: scm_libbase_internal.UnifiedChemicalSystem, atom: scm_libbase_internal.UnifiedAtom, region: str) -> None

Removes an atom from a region.

remove_atoms (*self:* UnifiedChemicalSystem *(page 83)*, *atom_indices: numpy.ndarray[numpy.int32[m, 1]]*) → None

Removes multiple atoms from the system, given their atom indices.

remove_atoms_from_region (*self:* UnifiedChemicalSystem *(page 83)*, *atom_indices: numpy.ndarray[numpy.int32[m, 1]]*, *region: str*) → None

Adds multiple atoms to a region, given their atom indices.

remove_region (*self:* UnifiedChemicalSystem *(page 83)*, *region: str*) → None

Removes a region. Atoms in that region will keep existing, but will not longer be part of that region.

reorder_atoms (*self:* UnifiedChemicalSystem *(page 83)*, *atom_order: List[int]*) → None

Reorders the system's atoms given a list of atom indices in the desired order.

static rmsd (*a:* UnifiedChemicalSystem *(page 83)*, *b:* UnifiedChemicalSystem *(page 83)*, *align: bool*, *unit: str = 'angstrom'*) → float

Computes the RMSD between two systems. 'align': whether or not the systems should be roto-translated as to maximize the RMSD.

rotate (*self:* UnifiedChemicalSystem *(page 83)*, *rot_mat: numpy.ndarray[numpy.float64[3, 3]]*) → None

Rotate the system according to the rotation matrix.

rotation_axis_for_angle_change (*self:* UnifiedChemicalSystem *(page 83)*, *arg0: int*, *arg1: int*, *arg2: int*) → numpy.ndarray[numpy.float64[3, 1]]

Determines the rotation axis for changing the angle between three atoms.

static rotation_matrix_minimizing_rmsd (*a:* UnifiedChemicalSystem *(page 83)*, *b:* UnifiedChemicalSystem *(page 83)*) → numpy.ndarray[numpy.float64[3, 3]]

Given two chemical systems, returns the rotation matrix that minimizes the RMSD.

select_all (*self:* UnifiedChemicalSystem *(page 83)*) → None

Selects all atoms.

select_atom (**args*, ***kwargs*)

Overloaded function.

1. select_atom(self: scm_libbase_internal.UnifiedChemicalSystem, arg0: int) -> None

Selects an atom given its index.

2. select_atom(self: scm_libbase_internal.UnifiedChemicalSystem, arg0: scm_libbase_internal.UnifiedAtom) -> None

Selects an instance of an atom.

select_atom_close_to_origin (*self:* UnifiedChemicalSystem *(page 83)*) → None

Selects the atom that is closest to the origin of the coordinate system.

select_atoms (*self:* UnifiedChemicalSystem *(page 83)*, *atom_indices: numpy.ndarray[numpy.int32[m, 1]]*) → None

Selects multiple atoms at once, given their atom indices.

select_atoms_if (*self:* UnifiedChemicalSystem *(page 83)*, *pred: Callable[[scm_libbase_internal.UnifiedAtom], bool]*) → None

Selects atoms based on a predicate function.

select_atoms_of_same_type (*self:* UnifiedChemicalSystem *(page 83)*) → None

Selects all atoms whose element is the same as of a currently selected atom.

select_connected (*self:* UnifiedChemicalSystem *(page 83)*) → None

Selects all atoms bonded to the currently selected atoms.

select_connected_if (*self:* UnifiedChemicalSystem *(page 83)*, *pred: Callable[[scm_libbase_internal.UnifiedAtom, scm_libbase_internal.UnifiedAtom, scm_libbase_internal.UnifiedBond], bool]*) → None

Selects atoms bonded to the currently selected atoms based on a predicate function.

select_molecule (*self:* UnifiedChemicalSystem *(page 83)*) → None

Using the bonds, selects entire molecules that include any currently selected atom.

select_molecule_if (*self:* UnifiedChemicalSystem *(page 83)*, *pred: Callable[[scm_libbase_internal.UnifiedAtom, scm_libbase_internal.UnifiedAtom, scm_libbase_internal.UnifiedBond], bool]*) → None

Using the bonds and a predicate function, selects all molecules that include a currently selected atom.

select_region (*self:* UnifiedChemicalSystem *(page 83)*, *region: str*) → None

Selects atoms in a region or region expression.

select_within_radius (*self:* UnifiedChemicalSystem *(page 83)*, *radius: float*, *unit: str = 'angstrom'*) → None

Selects all atoms within a given radius of any of the currently selected atoms.

set_angle (**args*, ***kwargs*)

Overloaded function.

1. set_angle(self: scm_libbase_internal.UnifiedChemicalSystem, a: int, b: int, c: int, angle: float, policy: scm_libbase_internal.UnifiedChemicalSystem.InternalCoordinateManipulationPolicy = <InternalCoordinateManipulationPolicy.SecondPartMoves: 1>) -> None

Sets the angle between three atoms by moving atoms according to a policy.

2. set_angle(self: scm_libbase_internal.UnifiedChemicalSystem, a: int, b: int, c: int, angle: float, unit: str, policy: scm_libbase_internal.UnifiedChemicalSystem.InternalCoordinateManipulationPolicy = <InternalCoordinateManipulationPolicy.SecondPartMoves: 1>) -> None

Sets the angle between three atoms by moving atoms according to a policy.

3. set_angle(self: scm_libbase_internal.UnifiedChemicalSystem, a: int, b: int, c: int, angle: float, rotation_axis: numpy.ndarray[numpy.float64[3, 1]], moving_atoms: numpy.ndarray[numpy.int32[m, 1]]) -> None

Sets the angle between three atoms by moving a specified set of atoms.

4. set_angle(self: scm_libbase_internal.UnifiedChemicalSystem, a: int, b: int, c: int, angle: float, unit: str, rotation_axis: numpy.ndarray[numpy.float64[3, 1]], moving_atoms: numpy.ndarray[numpy.int32[m, 1]]) -> None

Sets the angle between three atoms by moving a specified set of atoms.

set_atom (*self:* UnifiedChemicalSystem *(page 83)*, *atom_index: int*, *atom: scm_libbase_internal.UnifiedAtom*) → None

Safe assignment of an Atom instance to atoms[atom_index].

set_atoms_in_region (*self:* UnifiedChemicalSystem *(page 83)*, *atom_indices: numpy.ndarray[numpy.int32[m, 1]]*, *region: str*) → None

Creates or sets an entire region given the indices of the atoms to be part of the region.

set_density (*self:* UnifiedChemicalSystem *(page 83)*, *target_density: float*, *unit: str = 'dalton/angstrom3'*) → None

Applies a uniform strain to match the specified target density. Only valid for 3D periodic systems.

set_dihedral (**args*, ***kwargs*)

Overloaded function.

1. set_dihedral(self: scm_libbase_internal.UnifiedChemicalSystem, a: int, b: int, c: int, d: int, angle: float, policy: scm_libbase_internal.UnifiedChemicalSystem.InternalCoordinateManipulationPolicy = <InternalCoordinateManipulationPolicy.SecondPartMoves: 1>) -> None

Sets the dihedral between four atoms by moving atoms according to a policy.

2. set_dihedral(self: scm_libbase_internal.UnifiedChemicalSystem, a: int, b: int, c: int, d: int, angle: float, unit: str, policy: scm_libbase_internal.UnifiedChemicalSystem.InternalCoordinateManipulationPolicy = <InternalCoordinateManipulationPolicy.SecondPartMoves: 1>) -> None

Sets the dihedral between four atoms by moving atoms according to a policy.

3. set_dihedral(self: scm_libbase_internal.UnifiedChemicalSystem, a: int, b: int, c: int, d: int, angle: float, moving_atoms: numpy.ndarray[numpy.int32[m, 1]]) -> None

Sets the dihedral between four atoms by moving a specified set of atoms.

4. set_dihedral(self: scm_libbase_internal.UnifiedChemicalSystem, a: int, b: int, c: int, d: int, angle: float, unit: str, moving_atoms: numpy.ndarray[numpy.int32[m, 1]]) -> None

Sets the dihedral between four atoms by moving a specified set of atoms.

set_distance (**args*, ***kwargs*)

Overloaded function.

1. set_distance(self: scm_libbase_internal.UnifiedChemicalSystem, a: int, b: int, dist: float, policy: scm_libbase_internal.UnifiedChemicalSystem.InternalCoordinateManipulationPolicy = <InternalCoordinateManipulationPolicy.LighterPartMoves: 2>) -> None

Sets the distance between two atoms by moving atoms according to a policy.

2. set_distance(self: scm_libbase_internal.UnifiedChemicalSystem, a: int, b: int, dist: float, unit: str, policy: scm_libbase_internal.UnifiedChemicalSystem.InternalCoordinateManipulationPolicy = <InternalCoordinateManipulationPolicy.LighterPartMoves: 2>) -> None

Sets the distance between two atoms by moving atoms according to a policy.

3. set_distance(self: scm_libbase_internal.UnifiedChemicalSystem, a: int, b: int, dist: float, moving_atoms: numpy.ndarray[numpy.int32[m, 1]]) -> None

Sets the distance between two atoms by moving a specified set of atoms.

4. set_distance(self: scm_libbase_internal.UnifiedChemicalSystem, a: int, b: int, dist: float, unit: str, moving_atoms: numpy.ndarray[numpy.int32[m, 1]]) -> None

Sets the distance between two atoms by moving a specified set of atoms.

set_fractional_coordinates (*self:* UnifiedChemicalSystem *(page 83)*, *frac_coords: numpy.ndarray[numpy.float64[m, 3]]*, *unit: str = 'angstrom'*) → None

Sets the fractional coordinates of all atoms. In non-periodic directions the plain coordinates should be passed in the specified unit.

set_lattice_displacements_from_minimum_image_convention (*self:* UnifiedChemicalSystem *(page 83)*) → None

Applies the minimum image convention which sets the lattice_displacements for all bonds to be the shortest possible.

set_num_lattice_vectors (*self:* UnifiedChemicalSystem *(page 83)*, *nvec: int*) → None

Sets the number of lattice vectors.

set_selected_atoms (*self:* UnifiedChemicalSystem *(page 83)*, *atom_indices: numpy.ndarray[numpy.int32[m, 1]]*) → None

Sets the selection to the given atom indices. Any previous selection is cleared.

shortest_path_between (**args*, ***kwargs*)

Overloaded function.

1. shortest_path_between(self: scm_libbase_internal.UnifiedChemicalSystem, from_atom: int, to_atom: int, dist_func: Callable[[int, int, scm_libbase_internal.UnifiedBond], float]) -> Optional[List[int]]

Returns the shortest path between two atoms (using a custom distance function).

2. shortest_path_between(self: scm_libbase_internal.UnifiedChemicalSystem, from_atom: int, to_atom: int) -> Optional[List[int]]

Returns the shortest path between two atoms (measured in number of hops).

3. shortest_path_between(self: scm_libbase_internal.UnifiedChemicalSystem, from_atom: scm_libbase_internal.UnifiedAtom, to_atom: scm_libbase_internal.UnifiedAtom, dist_func: Callable[[int, int, scm_libbase_internal.UnifiedBond], float]) -> Optional[List[int]]

Returns the shortest path between two atoms (using a custom distance function).

4. shortest_path_between(self: scm_libbase_internal.UnifiedChemicalSystem, from_atom: scm_libbase_internal.UnifiedAtom, to_atom: scm_libbase_internal.UnifiedAtom) -> Optional[List[int]]

Returns the shortest path between two atoms (measured in number of hops).

shortest_path_length_between (**args*, ***kwargs*)

Overloaded function.

1. shortest_path_length_between(self: scm_libbase_internal.UnifiedChemicalSystem, from_atom: int, to_atom: int, dist_func: Callable[[int, int, scm_libbase_internal.UnifiedBond], float]) -> float

Returns the lengths of the shortest path between two atoms (measured using a custom distance function).

2. shortest_path_length_between(self: scm_libbase_internal.UnifiedChemicalSystem, from_atom: int, to_atom: int) -> float

Returns the length of the shortest path between two atoms (measured in number of hops).

3. shortest_path_length_between(self: scm_libbase_internal.UnifiedChemicalSystem, from_atom: scm_libbase_internal.UnifiedAtom, to_atom: scm_libbase_internal.UnifiedAtom, dist_func: Callable[[int, int, scm_libbase_internal.UnifiedBond], float]) -> float

Returns the lengths of the shortest path between two atoms (measured using a custom distance function).

4. shortest_path_length_between(self: scm_libbase_internal.UnifiedChemicalSystem, from_atom: scm_libbase_internal.UnifiedAtom, to_atom: scm_libbase_internal.UnifiedAtom) -> float

Returns the length of the shortest path between two atoms (measured in number of hops).

shortest_path_lengths_from (**args*, ***kwargs*)

Overloaded function.

1. shortest_path_lengths_from(self: scm_libbase_internal.UnifiedChemicalSystem, from_atom: int, dist_func: Callable[[int, int, scm_libbase_internal.UnifiedBond], float]) -> List[float]

Returns the lengths of all shortest paths from a source atom using a custom distance function.

2. shortest_path_lengths_from(self: scm_libbase_internal.UnifiedChemicalSystem, from_atom: int) -> List[float]

Returns the lengths (in number of hops) of all shortest paths from a source atom.

3. shortest_path_lengths_from(self: scm_libbase_internal.UnifiedChemicalSystem, from_atom: scm_libbase_internal.UnifiedAtom, dist_func: Callable[[int, int, scm_libbase_internal.UnifiedBond], float]) -> List[float]

Returns the lengths of all shortest paths from a source atom using a custom distance function.

4. shortest_path_lengths_from(self: scm_libbase_internal.UnifiedChemicalSystem, from_atom: scm_libbase_internal.UnifiedAtom) -> List[float]

Returns the lengths (in number of hops) of all shortest paths from a source atom.

sort_atoms (*self:* UnifiedChemicalSystem *(page 83)*, *comp: Callable[[scm_libbase_internal.UnifiedAtom, scm_libbase_internal.UnifiedAtom], bool]*) → None

Sorts the system's atoms according to a user defined comparator function.

sorted_atom_order (*self:* UnifiedChemicalSystem *(page 83)*, *comp: Callable[[scm_libbase_internal.UnifiedAtom, scm_libbase_internal.UnifiedAtom], bool]*) → List[int]

Returns the atom order based on a user defined comparator function as a list of atom indices.

split (*self:* UnifiedChemicalSystem *(page 83)*, *part_indices: numpy.ndarray[numpy.int32[m, 1]]*) → List[*UnifiedChemicalSystem* (page 83)]

Splits the system into parts and returns a list of these parts as separate systems.

split_into_molecules (*self:* UnifiedChemicalSystem *(page 83)*) → List[*UnifiedChemicalSystem* (page 83)]

Splits the system into individual molecules based on the connectivity between atoms.

symmetrize (*self:* UnifiedChemicalSystem *(page 83)*) → str

Symmetrizes a system, using either `symmetrize_molecule` or `symmetrize_cell`, depending on periodicity.

symmetrize_cell (*self:* UnifiedChemicalSystem *(page 83)*, *precision: float = 0.1*) → Tuple[int, str, str]

Symmetrize a 3D cell and return the symmetry symbols (number, international symbol, schoenflies symbol).

symmetrize_molecule (*self:* UnifiedChemicalSystem *(page 83)*, *tolerance: float = 0.05*) → str

Symmetrizes and reorients a molecule.

total_mass (*self:* UnifiedChemicalSystem *(page 83)*) → float

Total mass in atomic mass units (dalton).

translate (*self:* UnifiedChemicalSystem *(page 83)*, *shift: numpy.ndarray[numpy.float64[3, 1]]*, *unit: str = 'angstrom'*) → None

Translates all atoms in the ChemicalSystem by a vector.

write_in (*self:* UnifiedChemicalSystem *(page 83)*, *filename: str*) → None

Writes a ChemicalSystem to file as an AMS System block.

write_kf (**args*, ***kwargs*)

Overloaded function.

1. write_kf(self: scm_libbase_internal.UnifiedChemicalSystem, filename: str, section: str = 'Molecule', omode: scm_libbase_internal.KFFile.OpenMode = <OpenMode.Any: 0>) -> None

Writes a ChemicalSystem to a section on a KF file.

2. write_kf(self: scm_libbase_internal.UnifiedChemicalSystem, kf: scm_libbase_internal.KFFile, section: str = 'Molecule') -> None

Writes a ChemicalSystem to a section on a KF file.

write_xyz (*self:* UnifiedChemicalSystem *(page 83)*, *filename: str*, *extended_xyz_format: bool = True*) → None

Writes a ChemicalSystem to an extended XYZ file.

property adfprops

An optional list of all ADFProperties for all atoms.

property atoms

A list of all Atom instances that are part of this ChemicalSystem.

property bandprops

An optional list of all BANDProperties for all atoms.

property bonds

The bonds of the system.

property charge

The total charge of the system in in atomic units.

property coords: AngstromCoordsArray

The coordinates of the atoms in Angstrom

property dftbprops

An optional list of all DFTBProperties for all atoms.

property electrostatic_embedding

The electrostatic embedding of the system.

property forcefieldprops

An optional list of all ForcefieldProperties for all atoms.

property guiprops

An optional list of all GUIProperties for all atoms.

property lattice

The lattice of the system.

property num_atoms

The number of Atoms in the ChemicalSystem.

property num_regions

Returns the number of regions used in the system.

property qeprops

An optional list of all QEProperties for all atoms.

property reaxffprops

An optional list of all ReaxFFProperties for all atoms.

property region_names

Returns a list of the names of all regions in the system.

class InternalCoordinateManipulationPolicy

Members:

FirstPartMoves

SecondPartMoves

LighterPartMoves

HeavierPartMoves

__init__ (*self:* InternalCoordinateManipulationPolicy *(page 98)*, *value: int*) → None

property name

class UnifiedElement

property Z

The atomic number of the element.

property bond_guess_connectors

Number of connectors of an atom as used by the AMS bond guessing algorithm.

property bond_guess_electro_negative

Whether or not an element is considered electronegative by the AMS bond guessing algorithm.

property bond_guess_metallic

Whether or not an element is considered metallic by the AMS bond guessing algorithm.

property color

The color of the element used in the AMS graphical user interface.

property lone_pairs

The element's number of lone pairs as used by the AMS bond guessing algorithm.

property mass

The atomic mass (in Dalton) of the most abundant naturally occurring isotope of the element.

property radius

The covalent radius of the element in angstrom.

property symbol

The symbol of the element, e.g `C` for carbon.

class UnifiedElements

static from_atomic_number(*Z: int*) → *UnifiedElement* (page 98)

Returns an Element given its atomic number.

static from_symbol(*symbol: str*) → *UnifiedElement* (page 98)

Returns an Element given the element symbol as a string.

CHAPTER

FIVE

AKFREADER

The `AKFReader` Python class provides a convenient way to read annotated data from the binary KF files generated by AMS.

Below is an example demonstrating some of the `AKFReader` class's functionality

```
#!/usr/bin/env amspython

from scm.akfreader import AKFReader

# ===== Load a kf file.
akf = AKFReader("test.results/dftb.rkf")


# ===== Print all content in the kf file:
akf.print_all_variables()


# ===== Read a specific variable from file.
coordinates = akf.read("Molecule%Coords")
print("Coordinates:")
print(coordinates)

# Coordinates:
# [[-1.65367945e+00  4.59827802e+00 -9.59515212e-09]
#  [ 5.88355572e-01  2.78581237e+00  1.09863437e-08]
#  ...
#  [-3.26489723e+00  3.74833282e+00  9.66428930e-01]
#  [-2.21256221e+00  5.05007997e+00 -1.93285786e+00]]


# ====== Retrieve the description, units, and additional information for a variable:
description_dict = akf.description("Molecule%Coords")
print("'Molecule%Coords' description:")
print(description_dict)

# 'Molecule%Coords' description:
# {'_type': 'float_array',
#  '_shape': [3, 'nAtoms'],
#  '_comment': 'Coordinates of the nuclei (x,y,z)',
#  '_unit': 'bohr'}


# ====== Read another variable, and do a unit conversion:
energy_in_eV = akf.read("AMSResults%Energy", units="eV")
```

(continues on next page)

(continued from previous page)

```
print(f"Energy: {energy_in_eV} [eV]")

# Energy: -203.21954672148195 [eV]


# ====== Read all variables matching the pattern:
history_energies = akf.read("History%Energy(#)")
print(history_energies)

# [('History%Energy(1)', -5.758015498298242),
#  ('History%Energy(2)', -5.765039556660788),
#  ('History%Energy(3)', -5.766261662994689),
#  ('History%Energy(4)', -5.766288141072665)]
```

Note: The *KFFile* (page 80) Python class can also be used to read KF files. Key differences between the `AKFReader` and the `KFFile`:

- The `AKFReader` can provide 'extra information' on the content of the KF files: it can give descriptions of sections, variables, units, and automatically reshapes matrices to the proper size/shape.
- The `KFFile` can also be used to write KF files (while the `AKFReader` can only read).

See also:

A command line utility `$AMSBIN/akf` (described *here* (page 30)) offers an alternative way of using the `AKFReader` tool.

See also:

The list of possible variables for the various kf files can be found in the respective user manuals: ams.rkf, adf.rkf, band.rkf, dftb.rkf

API documentation:

class AKFReader(*path: str*, *kf_def_path=None*, *strict=False*, *verbose=False*)

A class to read AMS binary output files in KF format.

description(*name: str*) → Dict[str, Any]

Return the description of the key as a dictionary.

find(*name: str*) → list

Given the name of a variable with patterns, return the list of actual variables that match the query.

E.g. if name='History%Energy(#)', this might return: ['History%Energy(1)','History%Energy(2)']

print_all_variables(*include_hidden=False*, *print_metadata=True*)

Print the whole content of the akf file

print_variable(*name*, *print_metadata=True*)

Print the content and description of a variable

read(*name: str*, *units=None*)

Read and return the value of a variable.

In case that the name is a pattern like 'foo(#)' the result is a list of tuples for all vars matching the pattern:

'foo(#)' -> [('foo(1)', value1), ('foo(3)', value3)]

Partial patterns are not yet supported, but the idea would be the same:
'bar(#,#)' -> [('bar(1,1)', value11), ('bar(1,2)',value12), ('bar(2,1)',value21)] 'bar(1,#)' -> [('bar(1,1)', value11), ('bar(1,2)',value12)]

variables(*include_hidden=False*) → List[str]

Returns a list of all available variables on the file in the form "section%variable".

CHAPTER

SIX

FLEXMD

FlexMD is a python library for Flexible multi-scale Molecular Dynamics simulation, developed by Rosa E. Bulo, Christoph Jacob, Stefano Borini, Tao Jiang, Jelle Boereboom, Stanislav Simko and Hans van Schoot.

6.1 Basic philosophy and intended usage

We present a flexible python library for molecular dynamics, specialized in multi-scale simulations in a broad sense. At its core, the library interfaces the Atomistic Simulation Environment (ASE) [*1* (page 112)] molecular dynamics modules with a wide range of molecular mechanics and electronic structure codes. As such, it allows simple dynamics using forces computed with any energy/gradient evaluator provided by the ADF package.

Additionally, FlexMD allows the partitioning of a system into regions described at different resolution, with the aim of running multi-scale (hybrid) force calculations. Besides the traditional, rigid, multi-scale partitioning, FlexMD includes different schemes for Adaptive Multi-scale Molecular Dynamics. Such simulations allow the resolution of a particle to change according to its distance from a predefined active site, which is a necessity for successful multi-scale description of diffusive systems such as chemical reactions in solution.

Finally, the library couples the dynamics to rare events techniques, either implemented in FlexMD itself, or accessible through an interface with the PLUMED library for free energy calculations [*7* (page 113)], opening the possibility for evaluation on time-scales beyond the reach of standard molecular dynamics simulations.

The FlexMD package is designed to make simulation options possible that are not available natively in the ADF package. Its flexible nature makes it very versatile, but comes at a cost. This cost might be completely negligible in most simulations, but it can be very high in some cases (usually when combining only cheap methods such as forcefields).

The intended users for the FlexMD package are those with some Unix/Linux experience and a basic understanding of the Python Programming Language (http://python.org/). The user is also supposed to have a basic understanding of the various methods he wishes to combine. For example, if metadynamics is supposed to be combined with ADF, FlexMD expects the user to have knowledge about DFT calculations and the usage of Collective Variables. Finally, as with every computational method, the user should monitor the FlexMD performance, both in accuracy and speed.

6.2 FlexMD functionality summary

Molecule

Input/output

- Reads and writes PDB and XYZ files
- Reads and writes topology data (in CHARMM format)
- Reads and writes force field data (on CHARMM format)

Analysis

- Extracts geometry data

Drawing functionality

- Adds atoms and bonds
- Changes bond-lengths, angles and torsions
- Cuts fragments
- Cuts solvent boxes and droplets
- Performs rotations and translations, to fit bonds to axes and planes

Periodic functionality

- Adds periodic images
- Wraps molecules into periodic box

Water specific

- Finds hydrogen bonds
- Finds shortest water bridge connecting H-donor and acceptor

Energy and force calculations

Standard

- ADF
- DFTB
- REAXFF
- UFF
- MOPAC
- NAMD
- Lennard-Jones force fields

Multi-scale

- QM/MM, mechanical embedding: Combines all the codes above
- Hybrid: More flexible than QM/MM. Combines different force calculations by summing or subtracting the energies and forces. The standard calculations (above) can therefore be combined with:
  - Metadynamics
  - Plumed (external code that computes free energy data)
  - Constraints
- Adaptive QM/MM (for chemistry in solution)
  - Difference-based Adaptive Solvation (DAS)
  - Sorted Adaptive Partitioning (SAP)
  - Buffered-Core (BC)
  - Flexible Inner Region Ensemble Separator (FIRES)

Molecular Dynamics

- Uses ASE as the molecular dynamics driver for all above methods
- Analyses trajectories

6.3 Introduction

FlexMD is a python package providing molecular dynamics (MD) simulations using the energy evaluation methods made available by the ADF suite. A set of example scripts can be found in the examples/scmlib directory of a standard ADF installation.

FlexMD can be accessed interactively by running amspython, followed by a standard python import command for the package scm.flexmd. The python help function can be used to obtain detailed documentation about all FlexMD classes. In the following example, an inquiry of one class (the MDMolecule class) can be performed.

```
$ amspython
 from scm import flexmd
 help(flexmd.MDMolecule)
```

To leave the interactive help, press q. The help function can also be used to list the contents of the FlexMD package:

```
$ amspython
 from scm import flexmd
 help(flexmd)
```

Python can also give the help documentation as plain text:

```
$ amspython
 from scm import flexmd
 import pydoc
 print pydoc.render_doc(flexmd.ForceJob, "Help on %s")
```

6.4 Molecular Dynamics

FlexMD defines the molecular system under study through the **MDMolecule** class: an instantiation of this class holds all information about the molecular system to be simulated, such as coordinates, topology, and force field parameters (if needed). An **MDMolecule** object can be initialized from a PDB or XYZ file, by specifying its path at object creation.

An interface to energy evaluators is provided by specialized **ForceJob** classes, acting as wrappers around the ADF suite of programs. A **ForceJob** requires an **MDMolecule** object to be specified at creation. The resulting **ForceJob** object can either be used directly by the Atomistic Simulation Environment (ASE) [*1* (page 112)] library as a calculator object (see examples/scmlib/ASE_emt_h2o) or with the **ASEMDPropagator** class, which provides methods for running an MD time step using ASE classes. Internally, the propagator sets up the required ASE objects, passes the **ForceJob** object to them, and retrieves the new positions and velocities. An additional protagonist, an **MDManager** class instance, coordinates the MD simulation by running the MD steps with the **ASEMDPropagator** object and writing trajectory information to file.

During creation of an **MDManager** object, a directory 'QMMD' is created, which contains a file TRAJEC00.DCD holding the geometries along the trajectory, a file FTRAJEC00.DCD holding the forces along the trajectory, and finally a file ENERGY00.dat holding the potential and kinetic energy, as well as the temperature throughout the evaluation. To extract the geometries from the trajectory file, the **DCDFile** class is available, providing methods to read and write geometries to and from a trajectory file in DCD format. The **MDManager** is also responsible for handling restart of a previous MD evaluation: if a 'QMMD' directory is already present at script invocation, the new output files will be assigned the number subsequent to the highest numbered files in that directory. In addition, provided the previous run terminated normally, the restart will continue from the final geometry and velocity of the previous run.

The ADF package contains different electronic structure methods of varying degrees of accuracy and speed. The best-known methods are the ADF Density Functional Theory (DFT) code itself, and the BAND DFT code for periodic systems. FlexMD provides an interface toward both programs. For the interface with ADF, FlexMD makes use of classes from PyADF [*2* (page 112)], a scripting framework for efficient quantum chemistry calculations. In addition to ADF and BAND, several semi-empirical methods are included in the ADF suite, such as DFTB and the NDDO type schemes available in the MOPAC package [*3* (page 112)]. The ADF suite also provides classical mechanics methods, such as the reactive force field ReaxFF and the simple force field UFF. Interfaces to all of these methods are available in FlexMD. A simple example of a python script for MD using the **UFFForceJob** class for UFF calculations can be found in the examples directory, under examples/scmlib/flexmd_uff_h2o.

To increase the flexibility of FlexMD, an interface towards force calculations using the NAMD2.8 classical molecular dynamics package is provided (examples/scmlib/namd_h2o). NAMD2.8 is not distributed with the ADF suite, but it is available from a third party to be downloaded and installed (http://www.ks.uiuc.edu/Development/Download/download.cgi).

6.5 Multi-scale Molecular Dynamics

The design of the **ForceJob** class allows for flexible extension of its behavior, while at the same time keeping the client code unaware of its nature: it can either act as a simple wrapper for ADF programs, or it can be a more complex orchestrating class, combining simpler **ForceJob** classes to implement multi-scale strategies. One application of this extensible design can be found in the **QMMMForceJob** object, which combines a QM and an MM method in an IMOMM-type scheme (mechanical embedding). The **QMMMForceJob** object is assigned two other **ForceJob** objects, the first representing the high-resolution calculation (QM), while the other represents the low resolution (MM). Both **ForceJob** objects contain an **MDMolecule** object for the full molecular system. The selection of the QM-region is handled by the **QMMMForceJob**, which contains the information about the part of the molecule that constitutes the QM region. When forces are requested from the **QMMMForceJob**, the following behavior is orchestrated: first, a MM force calculation is performed on the full system; then, the QM-region is selected, a QM calculation is executed solely for that region, and energy and forces are added to those from the full system MM calculation. Finally, an MM calculation is computed for the small QM-region, and the energy and forces are subtracted, yielding the final result, returned to the invoker. In symbols:

EQM/MM(Full) = EMM(Full) + EQM(QMRegion) – EMM(QMRegion)

The **QMMMForceJob** handles periodic boundary conditions if the low-level (MM) method supports this feature (i.e. NAMD). Whether the periodic interaction of the QM region with itself is handled at high or low resolution depends on the method used for the QM calculation. An example of QM/MM MD calculations can be found in the examples directory examples/scmlib/qmmm_dftbUFF_2h2o. The **QMMMForceJob** allows the use of link atoms when the QM boundary cuts through covalent bonds. However, this feature comes at the price of an increased script complexity. An example of a QMMM link-atom MD simulation is provided in the examples directory, under examples/scmlib/qmmm_linkatom_dftbNAMD_glutamate.

For more complex multi-scale calculations the **HybridForceJob** class can be used. This class allows the combination of a large set of different **ForceJob**s, each of them describing either the same, or different molecular systems. Each **ForceJob** can either involve a calculation on the full **MDMolecule** object it contains, or restricted to a specified region of the corresponding molecule. The forces from each contributing **ForceJob** can either be added or subtracted from the total force according to user preference, as specified at construction of the **HybridForceJob** object.

In order to perform QM/MM simulations on chemical reactivity in solution, it is important that the description of the solvent molecules can change on the fly, as the molecules move towards or away from the reactive region. To facilitate this, an **AdaptiveQMMMForceJob** class is available to provide adaptive QM/MM simulations using several available schemes, as described by Bulo et al.[*4* (page 112)] and P. Fleurat-Lessard et al.[*8* (page 113)] In these schemes, the description of the diffusing molecules changes gradually from QM to MM and vice versa, based on the distance of those molecules to a predefined reactive site. Various schemes are available for assigning the QM and MM character of the molecules. The class contains a **QMMMForceJob** object, as well as a partitioning object that assigns the partial QM and MM character to the molecules. An examples python script for such an adaptive QM/MM simulation, using the DAS [*4* (page 112)] method, is provided in the examples directory, examples/scmlib/adqmmm_mopacscmUFF_h2o.

6.6 Biased Molecular Dynamics

Constraints can be added to a simulation using the derived **ForceJob** class **WallJob**. The constraint is in the form of a large one-dimensional Gaussian on the potential energy surface, along a predefined Collective Variable (CV). Examples of CV's are the distance between two atoms, the coordination number of two atoms, but also more complex quantities such as the minimum distance between two sets of atoms, or the distance of an atom to a hydroxide ion. The Collective Variables can be specified through the **CollectiveVariable** class. Derived **CollectiveVariable** classes are available to specify sums or multiples of other **CollectiveVariable** objects.

Regular MD calculations are limited in the time-scales achievable with current hardware. The order of such time-scales is much smaller than what is required for chemical reactions. To overcome this problem, two rare-events methods have been implemented directly into the library: metadynamics [*5* (page 112)] and umbrella sampling [*6* (page 113)]. Both these methods involve biasing the simulations along a CV. An example of a metadynamics input can be found in the examples directory in examples/scmlib/metadynamics_emt_h2o.

For a wider range of rare-events methods, FlexMD also offers an interface with the PLUMED library for free energy calculations [*7* (page 113)].To use this, a PLUMED input file is required, and for this we refer to the PLUMED manual. An example of a FlexMD input script using PLUMED can be found in the examples directory in examples/scmlib/plumed_emt_h2o.

6.7 Working with FlexMD

It is recommended to read the sections *Introduction* (page 107) and *Molecular Dynamics* (page 107) before working with FlexMD. Basic understanding of the Python Programming Language (http://python.org/) is also required. The Python website hosts documentation and a tutorial (http://docs.python.org/2/tutorial/) that can be used to learn Python.

The performance of the FlexMD package is difficult to predict because it depends on system size, the type of ForceJobs used and how these ForceJobs are combined. It is advised to first test the overhead of the FlexMD package for your system before running large simulations. When ab initio forces are involved, the overhead should not give a significant performance penalty. However, it may become a bottleneck when your system only uses cheap forcefields.

6.7.1 Creating a molecule object

FlexMD can be run trough the interactive python interpreter in the ADF package. To start it, run: $AMSBIN/amspython in a terminal, followed by:

```
from scm import flexmd
```

Note that it is also possible, and usually more convenient, to write your FlexMD code in a file and then to execute this file. To do this, type all the commands you would use in the interactive interpreter in a file, and then enter $AMSBIN/amspython myFlexMDjob.py in a terminal (after changing to the directory where the file was stored of course).

Most FlexMD jobs will start with importing FlexMD and creating an MDMolecule object. This can be done by starting from a geometry in xyz or pdb format, or by manually adding the atoms in the FlexMDjob.py file. Geometries can be generated in the ADF GUI, and then be exported to xyz file. For more details on the MDMolecule object, run $AMSBIN/amspython, import flexmd and call help(flexmd.mdmolecule).

```
from scm import flexmd
myMol = flexmd.MDMolecule('myGeometryFile.xyz')
```

Some ForceJobs require the system to be periodic. If we create an MDMolecule object from a pdb file that includes periodic information, the periodic boundary conditions are automatically imported. If the information is not there, we can add it to the MDMolecule object:

```
myMol = flexmd.pdb.set_box([50.0,25.0,100.0])
```

Info on set_box (and other functions, such as set_cellvectors, and write_pdb) can be found using help(flexmd.pdbmolecule).

It is also possible to write the info in the MDMolecule object to a pdb file. to do so, call pdb.write_pdb('mypdbfile.pdb') on the myMol object:

```
myMol.pdb.write_pdb('mypdbfile.pdb', box=True)
```

6.7.2 Creating a ForceJob

To specify what type of forces we want to use in the MD simulation, a ForceJob must be created. FlexMD has a number of ForceJobs (see PACKAGE CONTENTS in help(flexmd)), most of them with examples in $AMSHOME/examples/scmlib. The ForceJobs can be combined into a single ForceJob using flexmd.hybrid_ForceJob. As an example, we combine a reaxff_ForceJob with a metadynamicsjob and a walljob:

```
from scm import flexmd
myMol = flexmd.MDMolecule('myGeometryFile.xyz')
myMol = flexmd.pdb.set_box([50.0,25.0,100.0])
   # setup our reaxff ForceJob and attach the forcefield file
   # (place the ff file in the same dir as the script and the xyz!)
myReaxffForceJob = flexmd.ReaxffForceJob(molecule=myMol)
myReaxffForceJob.settings.set_ff_filename('reax_forcefield_file.ff')

   # next we define the collective variable: the distance between atom 1 and 2
myCvs = [flexmd.DistCV([1,2])]
   # create a set of metadynamics properties, using the CV
mtdSettings = flexmd.MetadynamicsSettings(cvs=myCvs, widths=[0.30], height=0.25 )
   # create the metadynamics job by combining the molecule, settings (with CV)
   # and the number of md steps between depositing metadynamics hills.
myMetadynamicsjob = flexmd.MetadynamicsJob( myMol, settings=mtdSettings, nstep=150 )

   # add a wall to prevent the two atoms from drifting more than 10 Angstrom away.
myWalljob = flexmd.WallJob(molecule=myMol, cvs=myCvs, cntrs=[10.0], widths=[1.0],␣
↪heights=[500.0])

   # combine the forces into a hybrid job that will be used for the MD
myForceJob = flexmd.HybridForceJob( [[myReaxffForceJob,'+'], [theMetadynamicsjob,'+'],
↪ [theWalljob,'+']], myMol )
```

Note that all the ForceJobs require some special input and settings, and that these settings can be applied both before and after defining the ForceJob. For the reaxffForceJob, we first define the ForceJob, and add the forcefield parameters file afterwards. For the metadynamics job we reverse this, and first create a metadynamicsJobSettings object, which is then used in the creation of the metadynamics job. For more detailed info on the different ForceJobs and their inputs, see the help function by calling help on a ForceJob, for example: help(flexmd.ReaxffForceJob) or help(flexmd.WallJob). Also remember that other examples of ForceJobs can be found in $AMSHOME/examples/scmlib.

6.7.3 Creating and running the MD job

Before the simulation can be set in motion, a propagator is needed. The propagatorJob controls simulation settings such as temperature and timestep size. FlexMD uses the Atomistic Simulation Environment (ASE) [*1* (page 112)] for this. The MDPropagatorJob object is created just like the other objects in FlexMD:

```
   # do this after importing flexmd and creating a ForceJob.
   # it creates the MDPropagator job, with some settings
myMDJob = flexmd.ASEMDPropagatorJob( ForceJob=myForceJob )
myMDJob.settings.set_tempcontrol( True, nhfreq=2, maxdef=50.0 )
myMDJob.settings.set_temperature(300.0)
myMDJob.settings.set_timestep( 0.02 )
```

For more details on the ASEMDPropagatorJob, view it's help page: help(flexmd.ASEMDPropagatorJob), or take a look at the MDSettings object: help(flexmd.MDSettings).

The propagatorJob can be used to create an MDManager object:

```
   # create an MD manager
myManager = flexmd.MDManager( mdjob=myMDJob)
```

The manager object is now in control of the MD simulation, and we can use it to run the simulation for a number of steps:

```
   # tell the MD manager to run the simulation
myManager.run( ncycles = 2500 )
```

Note that the number of steps here should be increased a lot if metadynamics effects are to be observed, but it is always wise to first run a small number of steps to check if everything works. Some information will be printed during the simulation, depending on the settings of the components used. The manager will also create some folders in the working directory, and store the data produced by the simulation in there.

The full flexmd jobfile should now look something like this:

```
from scm import flexmd
myMol = flexmd.MDMolecule('myGeometryFile.xyz')
myMol = flexmd.pdb.set_box([50.0,25.0,100.0])
   # setup our reaxff ForceJob and attach the forcefield file
   # (place the ff file in the same dir as the script and the xyz!)
myReaxffForceJob = flexmd.ReaxffForceJob(molecule=myMol)
myReaxffForceJob.settings.set_ff_filename('reax_forcefield_file.ff')

   # next we define the collective variable: the distance between atom 1 and 2
myCvs = [flexmd.DistCV([1,2])]
   # create a set of metadynamics properties, using the CV
mtdSettings = flexmd.MetadynamicsSettings(cvs=myCvs, widths=[0.30], height=0.25 )
   # create the metadynamics job by combining the molecule, settings (with CV)
   # and the number of md steps between depositing metadynamics hills.
myMetadynamicsjob = flexmd.MetadynamicsJob( myMol, settings=mtdSettings, nstep=150 )

   # add a wall to prevent the two atoms from drifting more than 10 Angstrom away.
myWalljob = flexmd.WallJob(molecule=myMol, cvs=myCvs, cntrs=[10.0], widths=[1.0],␣
↪heights=[500.0])

   # combine the forces into a hybrid job that will be used for the MD
myForceJob = flexmd.HybridForceJob( [[myReaxffForceJob,'+'], [theMetadynamicsjob,'+'],
↪ [theWalljob,'+']], myMol )

   # do this after importing flexmd and creating a ForceJob.
```

(continues on next page)

(continued from previous page)

```
   # it creates the MDPropagator job, with some settings
myMDJob = flexmd.ASEMDPropagatorJob( ForceJob=myForceJob )
myMDJob.settings.set_tempcontrol( True, nhfreq=2, maxdef=50.0 )
myMDJob.settings.set_temperature(300.0)
myMDJob.settings.set_timestep( 0.02 )

   # create an MD manager
myManager = flexmd.MDManager( mdjob=myMDJob)
   # tell the MD manager to run the simulation
myManager.run( ncycles = 2500 )
```

This job can be resumed by just running the script again, FlexMD should pick up on the coordinates and velocities from the QMMD folder. To do this, simply run the job again, and a new set of trajectory and energy files should appear in the QMMD folder. If you wish to specify which files FlexMD restarts from, you can use the set_restartnum() function on myMDJob.settings:

```
# add this below the "myMDJob.settings.set_timestep( 0.02 )" line
   # FlexMD will restart from the files with index numbers 1 lower than you pass to␣
↪set_restartnum
   # This will cause FlexMD to restart from QMMD/TRAJEC00.dcd and QMMD/TRAJEC00.dcd
myMDJob.settings.set_restartnum(1)
```

6.8 Required Citations

When you publish results in the scientific literature that were obtained with programs of the ADF package, you are required to include references to the program package with the appropriate release number, and a few key publications.

For calculations with FlexMD:

1. FlexMD 2017, SCM, T. Jiang, H. van Schoot, S. Simko, C. R. Jacob, S. Borini, R. E. Bulo, *A python library for flexible multi-scale molecular dynamics simulations.* http://www.scm.com

6.8.1 External programs and Libraries

Third party software used in the 2025.1 version of the Amsterdam Modeling Suite can be found in the file titled "third-party-software.txt" in the root of your AMS installation.

6.9 References

1. S.R. Bahn, K.W. Jacobsen, *An object-oriented scripting interface to a legacy electronic structure code.* Comput. Sci. Engin. 4, 56-66 (2002) (https://doi.org/10.1109/5992.998641). The Atomistic Simulation Environment website and documentation (https://wiki.fysik.dtu.dk/ase/)
2. C.R. Jacob, S.M. Beyhan, R.E. Bulo, A. Severo Pereira Gomes, A.W. Gotz, K. Kiewisch, J. Sikkema, L. Visscher, *PyADF - A scripting framework for multiscale quantum chemistry.* J. Comput. Chem. 32, 2328-2338 (2011) (https://doi.org/10.1002/jcc.21810)
3. J.J.P. Stewart, *Optimization of parameters for semiempirical methods IV: extension of MNDO, AM1, and PM3 to more main group elements.*, J. Mol. Model. 10, 155-164 (2004) (https://doi.org/10.1007/s00894-004-0183-z)
4. R.E. Bulo, B. Ensing, J. Sikkema, L. Visscher, *Toward a Practical Method for Adaptive QM/MM Simulations.* J. Chem. Theory Comput. 5, 2212-2221 (2009) (https://doi.org/10.1021/ct900148e)

5. A. Laio, M. Parrinello, *Escaping free-energy minima.* Proc. Natl. Acad. Sci. USA., 99, 12562-12566 (2002) (https://doi.org/10.1073/pnas.202427399)

6. B. Roux, *The calculation of the potential of mean force using computer simulations.* Comput. Phys. Commun. 91, 275-282 (1995) (https://doi.org/10.1016/0010-4655(95)00053-I)

7. M. Bonomi, D. Branduardi, G. Bussi, C. Camilloni, D. Provasi, P. Raiteri, D. Donadio, F. Marinelli, F. Pietrucci, R.A. Broglia, M. Parrinello, *PLUMED: A portable plugin for free-energy calculations with molecular dynamics.* Comp. Phys. Comm. 180, 1961-1972 (2009) (https://doi.org/10.1016/j.cpc.2009.05.011). PLUMED website and documentation (http://www.plumed-code.org/)

8. P. Fleurat-Lessard, C. Michel, R. E. Bulo, *Energy extrapolation schemes for adaptive multi-scale molecular dynamics simulations.* J. Chem. Phys. 137, 174111 (2012) (https://doi.org/10.1063/1.4739743)

CHAPTER

SEVEN

AUTOGRAFS

The Automatic Topological Generator for Framework Structures (http://pubs.acs.org/doi/abs/10.1021/jp507643v) is a software for the design ang high throughput generation of framework structures such as Metal-Organic Frameworks (MOFs), Zeolites, Covalent-Organic Frameworks (COFs), or any product of reticular chemistry. The underlying library for chemical objects makes extensive use of the Atomic Simulation Environment shipped with the Amsterdam Modeling Suite.

AuToGraFS was developed in the group of Prof. Dr. Heine at Jacobs University Bremen by Dr. M. Addicoat and D. Coupry. While a console application is available (https://github.com/maddicoat/AuToGraFS) under a GNU LGPL license, the version shipped together with the Amsterdam Modeling Suite was heavily modified into a fully importable python library for easy scripting capabilities. The software was also fully integrated into the GUI, and some native functions were developed for added functionality in this particular environment. A database of building units often used in existing MOFs is also provided, to cover basic needs.

The rest of this section is dedicated to a documentation and a demonstration of the usage of both the scripting capabilities and GUI tools from the Automatic Topological Generator for Framework Structures.

7.1 General AuToGraFS Scripting concepts

AuToGraFS relies on the concept of underlying nets common to many structures. These nets, or topologies can be found in the RCSR (http://rcsr.anu.edu.au/) and EPINET (http://epinet.anu.edu.au/) databases. Once implemented, AuToGraFS will follow these blueprints to generate a framework from building units of compatible geometries. Since these geometries are ideal objects, and most chemical structures are distorted, an optimization step using UFF is necessary for the building units to "snap into place". Correct handling of bonding information and UFF atom types are crucial to the generation of correct structures.

7.1.1 Components of AuToGraFS

The Fragment class

This is a slightly modified version of the native ASE Atoms object, designed to hold essential data for UFF postprocessing. This englobes the uff atom types, and the bonding information.

```
from scm.autografs.fragment import Fragment
from ase import Atoms

line = Fragment(ase.Atoms("X2", positions=...),
                mmtypes=["H_", "H_"],
                bonds = [[0,0,1],[0,0,1],[1,1,0]],
                shape="linear",
```

(continues on next page)

(continued from previous page)

```
                unit=None,
                name="a_line_has_no_name")
```

Where mmtypes is a list of the UFF atom types symbols, ordered as the corresponding atoms in the structure, and bonds is a symmetric numpy array (https://docs.scipy.org/doc/numpy-1.10.0/reference/generated/numpy.array.html) of size equal to the number of atoms. The i,j elements in the bonds array are the bond order between atom i and atom j of the structure. These bond orders should follow the UFF conventions (1.5 for aromatic, 0.5 for coordination bonds, 0.001 for hydrogen bonds…). The shape holds information for AuToGraFS about the connectivity of the fragment. Each dummy atom, with symbol "X" in ASE, represents a point of connection between two fragments. In the above example, the shape is clearly "linear". This is used to check compatibility between a fragment and a given topology. The unit of a fragment is set internally by AuToGraFS and is of no interest here. The name of a fragment is used only for logging purposes. Individual building blocks of a framework are manipulated through the Fragment class. It is possible to export a fragment from the GUI using edit -> framework -> export fragment. If the UFF atom types are not specified, an automatic typing script will take care of it.

The Model class

This is a container class that holds in one place the topology of a framework and the correctly positioned fragments. Most of the postprocessing methods found in AuToGraFS are Model methods.

```
from scm.autografs import *

model = autografs.Model(...)

# view the framework in adfinput. set clean to False to keep dummy atoms.
model.view(clean=True, verbose=True)

# write the name.ams and name.run files
model.write(name="framework", clean=True, verbose=True)

# returns a Fragment, with correct bonding information and mmtypes
# if indices is True, it also returns the index of the corresponding fragment in the␣
↪model for each atom
atoms = model.get_atoms(self, clean=False, indices=False)

# returns a supercell of the framework as a Fragment
supercell = model * (2,2,2)

# deletes a fragment in the framework. if no index is given, the fragment will be␣
↪chosen at random.
# the defects will cap with hydrogen the empty spaces. returns a Fragment
# here, a linker at random will be deleted
model.insert_defect(indices=None, centers=0, linkers=1)

# flip a linear object
model.flip_fragment(index=the_fragment_index)

# rotates a linear object around its axis by angle in degrees
model.rotate_fragment(index=the_fragment_index, angle=85.0)

# Will add a functional group to the selected fragment on a random hydrogen
# if and only if the hydogen is connected to a carbon.
# for more precise handling, use the GUI for now.
model.functionalize_fragment(functional_group="NH2", index=the_fragment_index)
```

The Autografs class

This class is the builder in itself. Given a path to a database of building units in .inp format (exportable via the GUI), it will generate any valid framework from a topology name, a center and linker name or objects, and optionally a pillar name or object. The linkers, centers and pillars objects have to be passed as Fragment .

```
from scm.autografs import *
from random import choice

# create the generator and update the database of building units
mofgen = autografs.Autografs(refresh_database=True, verbose=True, path="path/to/my/
↪fragments/")

# choose a topology from the available one given no center or linkers
topologies = mofgen.get_available_topologies(center=None, linker=None)
topology = choice(topologies)

# choose a center in the list of available centers for the topology
centers = mofgen.get_available_centers(topology=topology, linker=None)
center = choice(centers)

# choose a linker in the list of available linkers for the topology and center
linkers = mofgen.get_available_linkers(topology=topology,  center=center)
linker = choice(linkers)

# modify the linker by changing all hydrogens to fluorine
linker_structure = mofgen.database[linker]
linker_symbols = linker_structure.get_chemical_symbols()
linker_structure.set_chemical_symbols(["F" if l=="H" else l for l in linker_symbols])

# generate the framework using multiprocessing tools and automatic scaling of unit␣
↪cell
framework = mofgen.make(lable="my_framework", topology=topology, center=center,␣
↪linker=linker, pillar=None, form_factor=None, parallel=True)

# visualize in adfinput
framework.view()
```

do not forget to run a quick UFF optimization after generation to optimize lattice and fragment into an optimal configuration.

7.1.2 About the databases of building units

AuToGraFS uses both a binary database format, and structures in the DemonNano file format from which the binary is compiled. To ask AuToGraFS to use a custom directory containing .inp files, simply pass the directory to the path argument when initializing the constructor. The ".inp" file format should be formatted like the following example.

```
Data: SBUtype = linker
Data: shape = linear
Data: name = benzene
GEOMETRY CARTESIAN
C       0.00000000     -1.39103633      0.00000000     MMTYPE=C_R   QMMM=MM BOND=2/1.
↪5:6/1.5:12/1.0
C      -1.20606758     -0.69440959      0.00000000     MMTYPE=C_R   QMMM=MM BOND=1/1.
↪5:3/1.5:8/1.0
C      -1.20606758      0.69440959     -0.00000000     MMTYPE=C_R   QMMM=MM BOND=2/1.
```

(continues on next page)

(continued from previous page)

```
↪5:4/1.5:10/1.0
C       -0.00000000      1.39103633      0.00000000     MMTYPE=C_R    QMMM=MM BOND=3/1.
↪5:5/1.5:11/1.0
C        1.20606758      0.69440959      0.00000000     MMTYPE=C_R    QMMM=MM BOND=4/1.
↪5:6/1.5:7/1.0
C        1.20606758     -0.69440959     -0.00000000     MMTYPE=C_R    QMMM=MM BOND=1/1.
↪5:5/1.5:9/1.0
H        2.02806392      1.16933783      0.00000000     MMTYPE=H_     QMMM=MM BOND=5/1.0
H       -2.02806392     -1.16933783      0.00000000     MMTYPE=H_     QMMM=MM BOND=2/1.0
H        2.02806392     -1.16933783      0.00000000     MMTYPE=H_     QMMM=MM BOND=6/1.0
H       -2.02806392      1.16933783      0.00000000     MMTYPE=H_     QMMM=MM BOND=3/1.0
X        0.00000000      2.09099413      0.00000000     MMTYPE=H_     QMMM=MM BOND=4/1.0
X        0.00000000     -2.09099413      0.00000000     MMTYPE=H_     QMMM=MM BOND=1/1.0
END
```

7.1.3 Using the overhauled Atom Typer

Both AuToGraFS and the GUI now use a new python library for the assignment of UFF types to atoms and generation of the bonding matrix. The process goes as follows:

```
from scm.autografs import atomtyper

# instantiate the typer on a readable molecule file (e.g: "mol.cif")
typer = atomtyper.MolTyper("path/of/fileToType")

# choose the UFF library. options are uff and uff4mof
library = read_db("uff4mof")

# actual typing is done here
typer.type_mol(library)

# prints the bond matrix as a numpy array
# item i, j of matrix is the bond order between atoms i and j
print typer.get_guibonds()

# print a list of UFF types in order
print typer.get_mmtypes()

# write a SCM-UFF input file with correct info
typer.structure.write(name="mytypedmol")
```

7.2 AuToGraFS Examples

The following examples are compatible with all calculators within the Amsterdam Modeling Suite. Any Python script utilizing these libraries should be executed using the "$AMSBIN/amspython" binary.

```
"$AMSBIN/amspython" code.py
```

These examples illustrate the construction of various metal-organic frameworks (MOFs) using their secondary building units and Reticular Chemistry Structure Resource (RCSR) topological codes. They are specifically designed to work with the version of AuToGraFS included in SCM.

7.2.1 Simplest approach

We will construct the structures of MOF-5 and UIO66 using two different approaches. In AuToGraFS, you have the option to build a MOF using secondary building units from the default database or by supplying your own building units. We will employ both approaches to empower users with greater autonomy in constructing their MOFs

1.1 Using default database

In SCM, there are two default databases available for AuToGraFS. The first database comprises approximately 82 secondary building units and serves as the default path for AuToGraFS. You can directly access this path from `amshome/atomicdata/autografs/database`. The following scripts offer the simplest method for constructing any MOF.

```
# MOF-5/IRMOF-1
from scm.autografs import autografs

# Initialize the model generator
mofgen = autografs.Autografs(sbu_path=None)

# The secondary building units should be provided as a lists and this lists.
sbu_names = ["Benzene_linear", "Zn_mof5_octahedral"]

topology_name = "pcu"
mof = mofgen.make(topology_name=topology_name, sbu_names=sbu_names)

# The MOF generated directly view using ASE visualizer as follows
mof.view()

# You can also choose to write the mof to a gin (GULP) file input file or any ASE␣
↪input format.

# Writing to .gin
mof.write("MOF-5", "gin")

# Writing to cif
mof.write("MOF-5", "cif")
```

Checking list of secondary building units in default database and topologies

You can quickly access the secondary building units in the default database with the following command: `mofgen.sbu`. Additionally, the topologies can be accessed as follows: `mofgen.topologies`.

1.2 Using same default database as the gui

The GUI utilizes an updated database containing 5483 secondary building units that are shipped with SCM. This data can be found in `~/.scm_gui/autografs/default` after the first time the GUI is used to build a MOF. The database can also be manually extracted from `$AMSHOME/atomicdata/autografs/database/database.tar.gz` into there. Custom built seconadry building units from the GUI are stored in `~/.scm_gui/autografs/custom`. We can leverage the GUI default database as follows:

```
import os

from scm.autografs import autografs

```

(continues on next page)

(continued from previous page)

```
sbu_path = os.path.join(os.path.expanduser("~"), ".scm_gui", "autografs", "default")
mofgen = autografs.Autografs(sbu_path=sbu_path)

sbu_names = ["Bis_phenylethynylbenzene_linear", "Zn_mof5_octahedral"]
topology_name = "acs"

mof = mofgen.make(topology_name=topology_name, sbu_names=sbu_names)
mof.view()

# UIO-66
sbu_names = ["UIO66_Zr_icosahedral", "Benzene_linear"]
topology_name = "fcu"

mof = mofgen.make(topology_name=topology_name, sbu_names=sbu_names)
mof.view()
mof.write("UIO66", "cif")
```

1.3 Finding suitable topologies for a given set of SBUs

One common problem often revolves around determining which topologies would be suitable for a given set of building units. This situation becomes particularly intriguing when you've synthesized a new MOF and possess knowledge of the building units used, but not sure of the precise structure. Identifying the range of available topologies can facilitate the construction of all potential MOFs stemming from these building units. The PXRD from these hypothetical structures can then be compared with the experimental diffraction patterns to infer the structure of the synthesized MOF. The following example provides a straightforward illustration, which can be adapted to more complex systems.

```
from scm.autografs import autografs

mofgen = autografs.Autografs()
print(mofgen.sbu)

sbu_names = ["Pyrene_linear", "Zn_mof5_octahedral"]

list_of_available_topologies = mofgen.list_available_topologies(from_list=sbu_names)
print(list_of_available_topologies)

for idx, topology_name in enumerate(list_of_available_topologies[:10]):
    try:
        mofgen.set_topology(topology_name=topology_name)
        print(topology_name)
        mof = mofgen.make(sbu_names=sbu_names)
        mof.view()
        mof.write(topology_name, "cif")
    except Exception:
        pass
```

1.4 Finding available building units for a given topology

Similarly, one may be interested in finding possible building units that can be used to construct a MOF with a specific topology. This becomes valuable when investigating stability or conducting studies on isoreticular MOFs, where one aims to analyze the properties of a given topology. In such cases, it is essential to construct hypothetical MOFs based on a specified topology.

```
from scm.autografs import autografs

mofgen = autografs.Autografs()

topology_name = "acs"

available_sbus = mofgen.list_available_sbu(topology_name=topology_name, coercion=True)

print(available_sbus)
# {
#     (0, 1): [
#         "Al_trimeric_prism",
#         "Benzene_hexagonal",
#         "H6BHEHPI_Yaghi_hexagonal",
#         "Persulfurated_benzene_hexagonal",
#         "ReS_cluster_octahedral",
#         "Zn_mof5_octahedral",
#         "Zn_octahedral_paddlewheel",
#     ],
#     (14, 15, 16, 17, 18, 19): [
#         "Acetylene_linear",
#         "Benzene_linear",
#         "Benzil_linear",
#         "Benzo_bis_dioxaborole_linear",
#         "Benzodithiophene_linear",
#         "Benzothiadiazole_linear",
#         "Betabinaphtol_linear",
#         "Bicyclooctane_linear",
#         "Biphenyl_linear",
#         "Bipyridine_linear",
#         "Bis_phenylethynylbenzene_linear",
#         "Butane_linear",
#         "Chrysene_linear",
#         "Cubane_linear",
#         "DCDPBN_Yaghi_linear",
#         "DPMDBDA_linear",
#         "Decapentene",
#         "Diethinylbenzene_linear",
#         "Diphenylbutadiyne_linear",
#         "Diphenylperylimide_linear",
#         "Fluorenone_linear",
#         "H4DH11PhDC_Yaghi_linear",
#         "H4DH9PhDC_Yaghi_linear",
#         "Naphtalene_linear",
#         "Pentaphenyl_linear",
#         "Peropyrene_linear",
#         "Phenazine_linear",
#         "Phenylethynylbenzene_linear",
#         "Pyrene_linear",
#         "Stilbene_linear",
#         "Tetracene_linear",
```

(continues on next page)

(continued from previous page)

```
#         "Tetraphenyl_linear",
#         "Thienothiophene_linear",
#         "Triphenyl_linear",
#         "Zn_porphyrin_linear",
#     ],
# }

selected_sbu = ["Al_trimeric_prism", "Acetylene_linear"]

mof = mofgen.make(topology_name=topology_name, sbu_names=selected_sbu)
mof.view()
```

N.B

There are a few important things to note regarding the following line of code: `available_sbus = molgen.list_available_sbu(topology_name=topology_name, coercion=True)` #### i. Coercion When setting coercion to True, AutoGrafs is compelled to match only SBUs that can combine based on their number of points of extension/ number of dummies/coordination number of SBUs. This feature is beneficial as it ensures a strong likelihood of building a MOF by combining outputs from `available_sbus`

ii. available_sbus

The `available_sbus` is a dictionary with keys represented as tuples. It's crucial to use items from different keys as centers and linkers. For instance:

```
available_sbus = {(0, 1): ['Al_trimeric_prism', 'Benzene_hexagonal', 'H6BHEHPI_Yaghi_
↪hexagonal', 'Persulfurated_benzene_hexagonal', 'ReS_cluster_octahedral', 'Zn_mof5_
↪octahedral', 'Zn_octahedral_paddlewheel'], (14, 15, 16, 17, 18, 19): ['Acetylene_
↪linear', 'Benzene_linear', 'Benzil_linear', 'Benzo_bis_dioxaborole_linear',
↪'Benzodithiophene_linear', 'Benzothiadiazole_linear', 'Betabinaphtol_linear',
↪'Bicyclooctane_linear', 'Biphenyl_linear', 'Bipyridine_linear', 'Bis_
↪phenylethynylbenzene_linear', 'Butane_linear', 'Chrysene_linear', 'Cubane_linear',
↪'DCDPBN_Yaghi_linear', 'DPMDBDA_linear', 'Decapentene', 'Diethinylbenzene_linear',
↪'Diphenylbutadiyne_linear', 'Diphenylperylimide_linear', 'Fluorenone_linear',
↪'H4DH11PhDC_Yaghi_linear', 'H4DH9PhDC_Yaghi_linear', 'Naphtalene_linear',
↪'Pentaphenyl_linear', 'Peropyrene_linear', 'Phenazine_linear',
↪'Phenylethynylbenzene_linear', 'Pyrene_linear', 'Stilbene_linear', 'Tetracene_linear
↪', 'Tetraphenyl_linear', 'Thienothiophene_linear', 'Triphenyl_linear', 'Zn_
↪porphyrin_linear']}
```

`key = (0, 1)` Items listed here represent the center nodes or centers.

`key = (14, 15, 16, 17, 18, 19)` Items listed here should be used as linkers.

For example, incorrect usage like:

`sbu_names ≠ ['Al_trimeric_prism', 'Benzene_hexagonal']` will cause the code to fail.

Instead, one should select items from different keys, such as:

`sbu_names = ['Al_trimeric_prism', 'Benzo_bis_dioxaborole_linear']`

or

`sbu_names = ['Benzene_hexagonal', 'DPMDBDA_linear']` note that this will build a COF instead of a MOF.

7.2.2 Making SURMOFs

Surface Mounted Metal-Organic Frameworks are a class of materials that combine organic linkers and metal ions or clusters to form highly porous structures. SURMOFs are unique because they can be grown as thin films directly on substrates, allowing for their integration into various devices and applications.

Simply put, these are pillared MOFs that are horizontally linked with a given linkered and vertically linked with another linker. The following examples can be used to generate all possible SURMOFs for a list of available sbus present in the default database. ### N.B Note that SURMOFs generally adopt the `pcu` topology.

```
from scm.autografs import autografs

mofgen = autografs.Autografs()

topology_name = "pcu"

# get all available linkers, center, pillars for the pcu topology
all_sbus = mofgen.list_available_sbu(topology_name=topology_name)

center_key, linker_key = list(all_sbus.keys())
centers = all_sbus[center_key]
linkers = all_sbus[linker_key]

for node in centers:
    for i in range(len(linkers) - 1):
        for j in range(1, len(linkers)):
            linker = linkers[i]
            pillar = linkers[j]
            if linker != pillar:
                sbu_names = [node, (linker, 1), (pillar, 1)]
                framework = mofgen.make(topology_name=topology_name, sbu_names=sbu_
↪names)
                surmof_name = "node-linker-pillar"
                framework.write(surmof_name, "gin")
```

INDEX

Non-alphabetical

A

B

C

D

E

F

G

H

I

R

S

T

U

V

W

X

Z