



Scripting Manual

Amsterdam Modeling Suite 2024.1

www.scm.com

Apr 05, 2024

CONTENTS

| | | |
|----------|---|-----------|
| 1 | Getting Started | 1 |
| 1.1 | Linux | 1 |
| 1.2 | MacOS | 1 |
| 1.3 | Windows | 1 |
| 1.4 | Notes on Python | 2 |
| 2 | Command Line Tools | 3 |
| 2.1 | AMSprep: generate (multiple) ADF jobs | 3 |
| 2.1.1 | Additional Notes | 7 |
| 2.2 | Tutorial: Generate structures for substituent effects screening | 7 |
| 2.3 | AMSreport: generate reports | 11 |
| 2.3.1 | Additional notes | 18 |
| 2.4 | KF command line utilities | 18 |
| 3 | Python Stack in Amsterdam Modeling Suite | 23 |
| 3.1 | General | 23 |
| 3.2 | Included modules | 23 |
| 3.3 | Using other modules with the AMS Python Stack | 24 |
| 3.4 | Install and run Jupyter Lab (Jupyter Notebooks) | 24 |
| 3.4.1 | Install Jupyter Lab | 24 |
| 3.4.2 | Run Jupyter Lab | 25 |
| 3.4.3 | Execute a notebook from the command-line | 25 |
| 3.5 | Python virtual environment | 25 |
| 3.5.1 | Default python virtual environment | 25 |
| 3.5.2 | Virtual environments for different AMS versions | 26 |
| 3.5.3 | Changing the location of the python virtual environment | 26 |
| 3.5.4 | Disabling the virtual environment | 26 |
| 3.5.5 | Uninstalling the virtual environment | 26 |
| 3.5.6 | Reinstalling the virtual environment | 26 |
| 3.6 | Useful commands | 26 |
| 4 | SCM libbase | 29 |
| 4.1 | Chemical System | 29 |
| 4.1.1 | Overview | 29 |
| 4.1.2 | Reading and writing | 30 |
| 4.1.3 | Atoms | 31 |
| 4.1.4 | Molecular properties | 33 |
| 4.1.5 | Geometry and manipulation | 34 |
| 4.1.6 | Bonds | 36 |
| 4.1.7 | Lattice and Periodic Systems | 40 |

| | | |
|----------|---|-----------|
| 4.1.8 | Regions | 42 |
| 4.1.9 | Atom selection | 45 |
| 4.1.10 | Atomic Properties | 47 |
| 4.1.11 | Converting to and from PLAMS Molecules | 49 |
| 4.2 | Units | 50 |
| 4.2.1 | List of available units | 51 |
| 4.3 | Input File | 58 |
| 4.4 | KFFile | 60 |
| 5 | ASE | 65 |
| 6 | FlexMD | 67 |
| 6.1 | Basic philosophy and intended usage | 67 |
| 6.2 | FlexMD functionality summary | 67 |
| 6.3 | Introduction | 69 |
| 6.4 | Molecular Dynamics | 69 |
| 6.5 | Multi-scale Molecular Dynamics | 70 |
| 6.6 | Biased Molecular Dynamics | 71 |
| 6.7 | Working with FlexMD | 71 |
| 6.7.1 | Creating a molecule object | 71 |
| 6.7.2 | Creating a ForceJob | 72 |
| 6.7.3 | Creating and running the MD job | 73 |
| 6.8 | Required Citations | 74 |
| 6.8.1 | External programs and Libraries | 74 |
| 6.9 | References | 74 |
| 7 | PLAMS | 77 |
| 7.1 | Required Citations | 77 |
| 7.1.1 | External programs and Libraries | 77 |
| 8 | AuToGraFS | 79 |
| 8.1 | General AuToGraFS Scripting concepts | 79 |
| 8.1.1 | Components of AuToGraFS | 79 |
| | The Fragment class | 79 |
| | The Model class | 80 |
| | The Autografs class | 81 |
| 8.1.2 | About the databases of building units | 81 |
| 8.1.3 | Using the overhauled Atom Typer | 82 |
| 8.2 | AuToGraFS Examples | 83 |
| 8.2.1 | Generation of all available pillared SURMOF | 83 |
| 8.2.2 | Generation of a defectuous UIO-66 MOF from custom files | 83 |
| 8.2.3 | Generation of conformers in the IRMOF-5 | 84 |
| 9 | AKFReader | 85 |
| | Index | 89 |

GETTING STARTED

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 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 yield the help message for the [amsprep](#) (page 3) command line tool:

```
amsprep -h
```

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`.

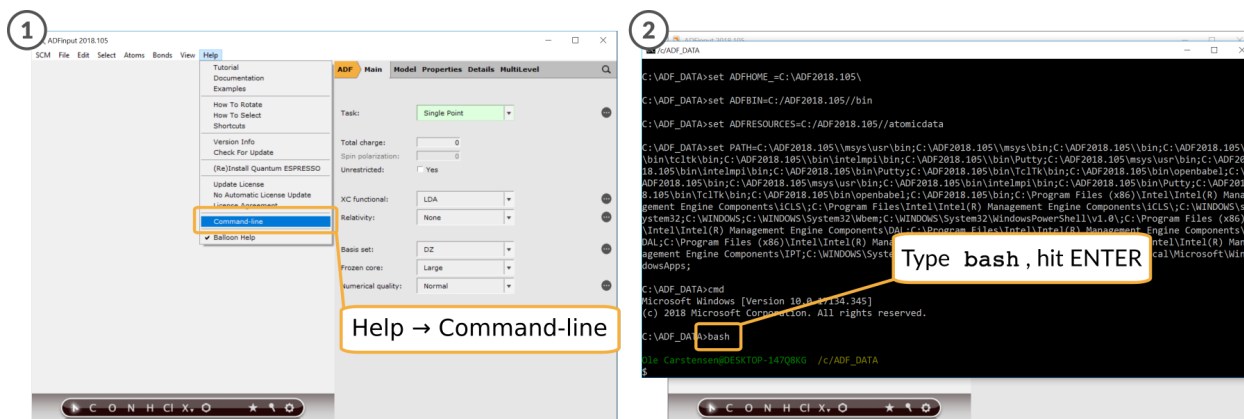
Alternatively, you can follow the Linux steps.

1.3 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`)

1.4 Notes on Python

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

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

Further information can be found here: *Python Stack in Amsterdam Modeling Suite* (page 23).

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.

2.1 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]
                                [-gradientonly]
                                [-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)

-gradientonly: 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/
↳Defaults.tpl

```

(continues on next page)

(continued from previous page)

```

    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

```

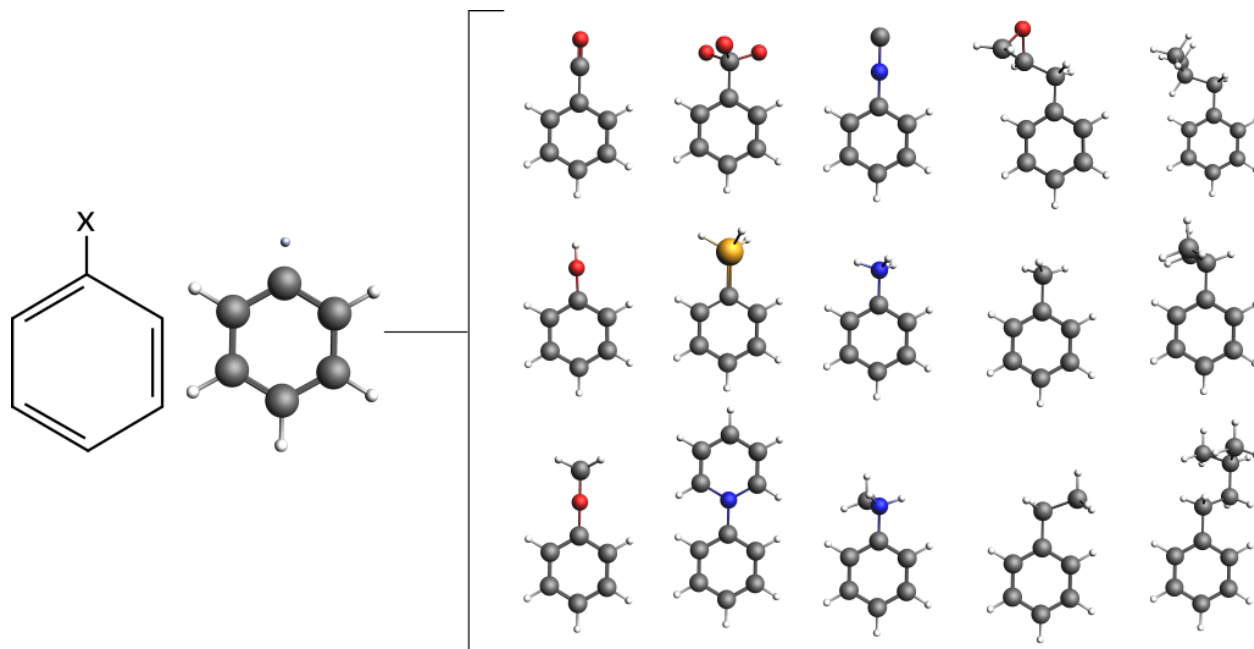
2.1.1 Additional Notes

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

2.2 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.

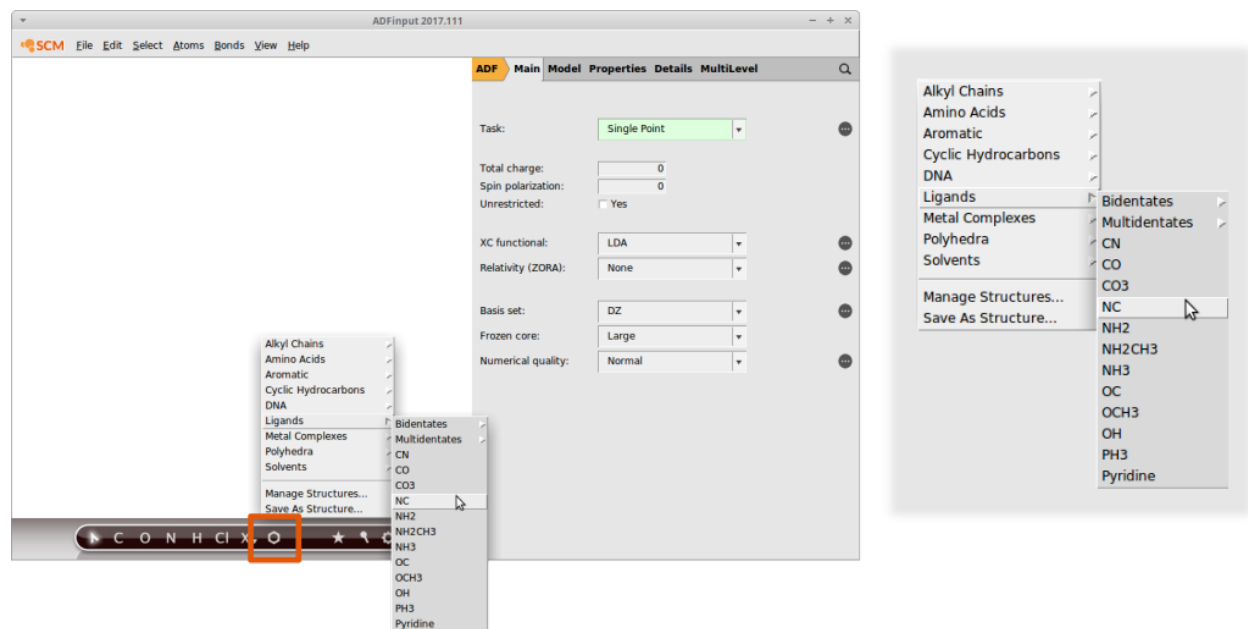


Contents:

- 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:

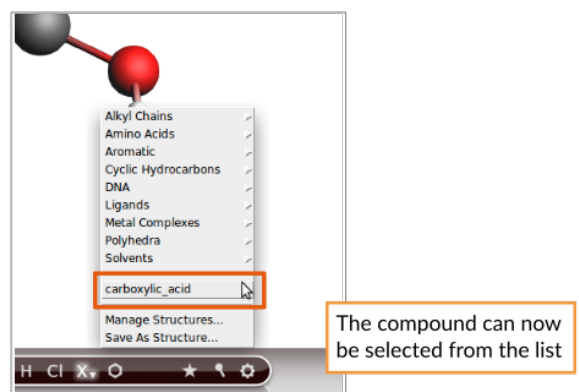
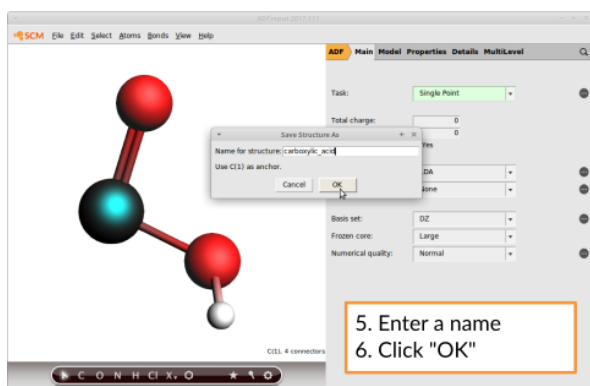
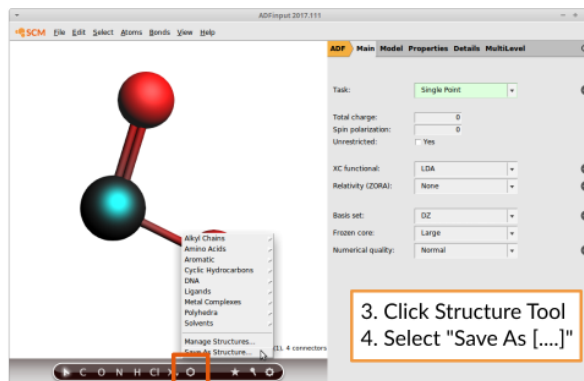
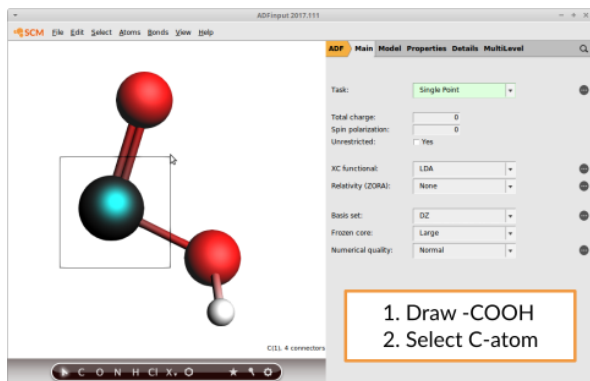


Note how the entries are organized. For example the isocyanide functional group (“NC”) can be found in “Ligands”.

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

as an anchor.

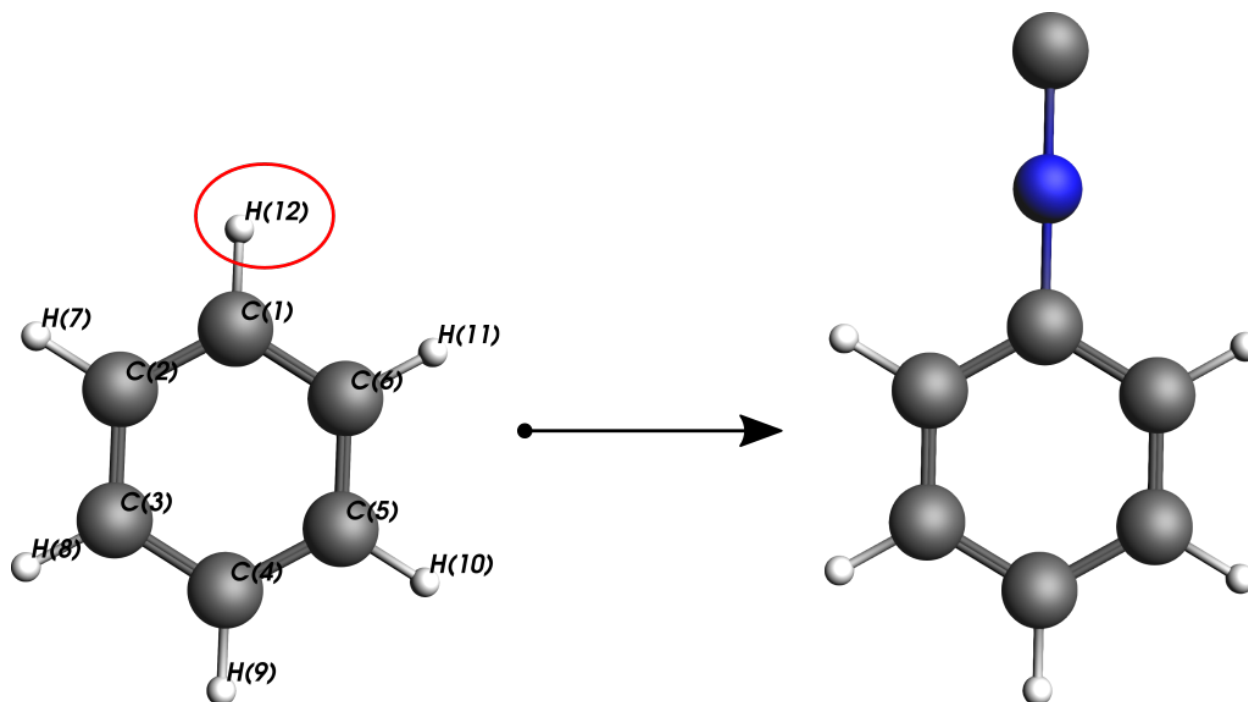
This is how you would add a carboxylic acid group:



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.

2.3 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 four 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.rxxf "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 (CONNECT 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
```

(continues on next page)

(continued from previous page)

```

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 #

2.3.1 Additional notes

- SDF and BGF records can be produced 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) (<http://www.scm.com/Downloads/KFReader-20140106.zip>) and can be modified for more specific needs.

2.4 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 be 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 of 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 udkmpkf 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 udkmpkf 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 85).

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
```

(continues on next page)

(continued from previous page)

```
-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.
-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
```


PYTHON STACK IN AMSTERDAM MODELING SUITE

3.1 General

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

| AMS version | Python version |
|-------------|----------------|
| AMS2024 | 3.8.12 |
| AMS2023 | 3.8.12 |
| AMS2022 | 3.6.9 |
| AMS2021 | 3.6.9 |
| AMS2020 | 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`.

3.2 Included modules

Some of the included modules are:

- `numpy` (1.21.2) and `scipy` (1.8.0): Big modules with a lot of functionality for math and science, more information on the [SciPy website](https://www.scipy.org/) (<https://www.scipy.org/>).
- `IPython` (7.22.0): An improved interactive python shell, more information can be found on the [iPython website](https://ipython.org/) (<https://ipython.org/>). Can be started with:

```
$AMSBIN/amsipython
```

- `ase` (3.22.1): ASE (Atomic Simulation Environment) is a python module for atomistic simulations, more information in the [ASE documentation](#) (page 65).
- `matplotlib` (3.5.1): A library for plotting data in 2D, more information on the [Matplotlib website](https://matplotlib.org/) (<https://matplotlib.org/>). We do not ship an interactive backend for matplotlib, so make sure to set a [non-interactive backend](https://matplotlib.org/faq/howto_faq.html#generate-images-without-having-a-window-appear) (https://matplotlib.org/faq/howto_faq.html#generate-images-without-having-a-window-appear) when using it. For example the Agg backend for PNGs:

```
import matplotlib
matplotlib.use('Agg')
```

- `pip` (21.3.1): The recommended tool for installing packages from the Python Package Index (PyPI). The [pip documentation](https://pip.pypa.io/en/stable/) (<https://pip.pypa.io/en/stable/>) explains in detail how to use this tool, but for the Python stack shipped with the Amsterdam Modeling Suite 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
```

- `flexmd`: A module for running MD simulations with adaptive QM/MM regions. Details can be found in the [FlexMD documentation](#) (page 67).
- `plams`: PLAMS (Python Library for Automating Molecular Simulation) is a collection of tools that aim at providing powerful, flexible and easily extendable Python interface to molecular modeling programs. It takes care of input preparation, job execution, file management and output processing as well as helps with building more advanced data workflows. See the [PLAMS tutorials](#) and [PLAMS documentation](#) for more information.
- `autografs`: AuToGraFS stands for Automatic Topological Generator for Framework Structures. Information and examples can be found in the [AuToGraFS documentation](#) (page 79).

3.3 Using other modules with the AMS Python Stack

You can extend the the AMS Python Stack with other modules. You can use `pip` (see above) to install additional modules if they are available on the Python Package Index (PyPI (<https://pypi.python.org/pypi>)). This is the recommended way of installing packages:

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

You can also do this for your own packages that you develop, for example:

```
$AMSBIN/amspython -m pip install -e .
```

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.

3.4 Install and run Jupyter Lab (Jupyter Notebooks)

3.4.1 Install Jupyter Lab

AMS2023 or later: Install Jupyter lab in the [Package Manager](#) (GUI: SCM → Packages, Command-line: `$AMSBIN/amspackages gui` or `$AMSBIN/amspackages -h`).

3.4.2 Run Jupyter Lab

In a 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)**.

3.4.3 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`.

3.5 Python virtual environment

3.5.1 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`.

3.5.2 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.

3.5.3 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.

3.5.4 Disabling the virtual environment

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

3.5.5 Uninstalling the virtual environment

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

3.5.6 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.

3.6 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
```


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.

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 coincide with it's center of mass
mol.translate(-mol.center_of_mass())

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

4.1.2 Reading and writing

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](#).

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 Serialization

When you read or write a `ChemicalSystem` using either the **System Block** format or a *kf file* (page 60), 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.

Available Methods for Serialization

Below are the methods you can use for reading and writing (also known as serializing and deserializing) a `ChemicalSystem`:

class `UnifiedChemicalSystem`

class `UnifiedChemicalSystem` (*system_block: str*)

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

classmethod `from_in` (*filename: str, name: str = ""*) → `UnifiedChemicalSystem`

Constructs and returns a new `ChemicalSystem` from a (possibly named) System block in an AMS input file.

classmethod `from_kf` (*filename: str, section: str = 'Molecule'*) → `UnifiedChemicalSystem`

classmethod `from_kf` (*kf: libbase.KFFile (page 60), section: str = 'Molecule'*) → `UnifiedChemicalSystem`

Constructs and returns a new `ChemicalSystem` from a section on a KF file.

classmethod `from_xyz` (*filename: str*) → `UnifiedChemicalSystem`

Constructs and returns a new `ChemicalSystem` from an extended XYZ file.

classmethod `from_input` (*input_file: libbase.InputFile (page 58), prefix: str*) → `UnifiedChemicalSystem`

Constructs and returns a new `ChemicalSystem` from an `InputFile` instance, given a prefix, e.g. 'System[1]%'.

write_in (*filename: str*) → `None`

Writes a `ChemicalSystem` to file as an AMS System block.

write_kf (*filename: str, section: str = 'Molecule', omode: libbase.KFFile.OpenMode (page 60) = KFFile.OpenMode.Any*) → None

write_kf (*kf: libbase.KFFile (page 60), section: str = 'Molecule'*) → None
Writes a ChemicalSystem to a section on a KF file.

write_xyz (*filename: str, extended_xyz_format: bool = True*) → None
Writes a ChemicalSystem to an XYZ file.

By default the file is written in the AMS extended XYZ file format, which includes lattice vectors, atomic properties, regions and some system properties such as the total charge. In order to write a plain standard XYZ files without any of these extensions, set `extended_xyz_format` to `False`.

4.1.3 Atoms

The ChemicalSystem class maintains an array of instances of the Atom class (UnifiedAtom) 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

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

property coords

The coordinates of the atoms in bohr.

contains_atom (*atom: libbase.UnifiedAtom*) → bool

Checks if an Atom instance is part of a ChemicalSystem.

atom_index (*atom: libbase.UnifiedAtom*) → int

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

add_atom (*atom*: libbase.UnifiedAtom) → None
add_atom (*atom*: libbase.UnifiedAtom, *coords*: ArrayLike, *unit*: str = 'bohr') → None
add_atom (*Z*: int, *coords*: ArrayLike, *unit*: str = 'bohr') → None
add_atom (*element*: libbase.UnifiedElement, *coords*: ArrayLike, *unit*: str = 'bohr') → None
add_atom (*symbol*: UnifiedAtom.T_AtomSymbol, *coords*: ArrayLike, *unit*: str = 'bohr') → 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*: libbase.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 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) → 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 the same lattice or no lattice at all. 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) → UnifiedChemicalSystem
 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) → UnifiedChemicalSystem
 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

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: Sequence[*numpy.int64*]*) → List[UnifiedChemicalSystem]

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*)

class UnifiedAtom (*symbol: T_AtomSymbol*)

Class representing a single atom and its properties.

property Z

The atomic number of the atom.

property coords

The coordinates of the atom in bohr.

property symbol

The element symbol, e.g. 'C' or 'Au'.

in_chemicalsystem () → bool

Check if this Atom instance is part of a ChemicalSystem.

4.1.4 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

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')

4.1.5 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
```

Available Methods

class UnifiedChemicalSystem

class UnifiedChemicalSystem(*system_block: str*)

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

translate(*shift: ArrayLike*) → 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*) → 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) (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*, *b*: *UnifiedChemicalSystem*) → *numpy.ndarray*[*Any*, *numpy.dtype*[*numpy.float64*]]

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

classmethod rmsd (*a*: *UnifiedChemicalSystem*, *b*: *UnifiedChemicalSystem*, *align*: *bool*) → 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 () → *numpy.ndarray*[*Any*, *numpy.dtype*[*numpy.float64*]]

Position of the geometric center in bohr.

center_of_mass () → *numpy.ndarray*[*Any*, *numpy.dtype*[*numpy.float64*]]

Position of the center of mass in bohr.

inertia_tensor () → *numpy.ndarray*[*Any*, *numpy.dtype*[*numpy.float64*]]

Returns the system's inertia tensor as a 3x3 matrix in atomic units ($\text{amu} \cdot \text{bohr}^2$).

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

moments_of_inertia () → *Tuple*[*numpy.ndarray*[*Any*, *numpy.dtype*[*numpy.float64*]], *numpy.ndarray*[*Any*, *numpy.dtype*[*numpy.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 in atomic units ($\text{amu} \cdot \text{bohr}^2$) sorted by magnitude
- a 3x3 *np.array* with 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* = 'bohr') → 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).

4.1.6 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
    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.

```
property lattice_displacements
```

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
```

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

The Bonds Class

This contains the information all the bonds, and methods to retrieve and manipulate bonds:

```
class UnifiedBonds
```

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

```
add_bond (from_atom: int, to_atom: int, bond: libbase.UnifiedBond (page 37)) → None
```

Adds a bond between two atoms, given their indices.

```
add_bonds (from_atoms: ArrayLike, to_atoms: ArrayLike, bonds: List[libbase.UnifiedBond (page 37)])  
→ 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.

```
num_bonds () → int
```

```
num_bonds (atom: int) → int
```

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

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

Checks whether two atoms are bonded or not.

```
get_bonds_for_atom (from_atom: int) → Iterator[Tuple[int, int, libbase.UnifiedBond (page 37)]]
```

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 $A \leq B$.

```
remove_bond (bidx: Union[int, libbase.UnifiedBond (page 37)]) → None
```

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

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

Removes the bonds between two atoms, given their indices.

```
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.

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

The bonds of the system.

guess_bonds () → None

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

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 () → numpy.ndarray[Any, numpy.dtype[numpy.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]

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: libbase.UnifiedAtom, to_atom: libbase.UnifiedAtom) → bool

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

atom_is_in_ring (atom: Union[int, libbase.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.

property molgraph_edgweight_func_t

A method (Callable[[int, int, UnifiedBond], float]) taking two atom indices and a Bond instance between the two atoms. It shall return the non-negative distance measurement between the two atoms. This is an (often optional) input to methods that determine the shortest path between atoms in the molecular graph.

molgraph_dijkstra (from_atidx: int, dist_func: molgraph_edgweight_func_t (page 38), to_atidx: int = -1) → Tuple[List[float], List[int], List[bool]]

General method implementing the Dijkstra algorithm on the molecular graph. The Dijkstra 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_edgweight_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*: Union[int, libbase.UnifiedAtom]) → List[float]
shortest_path_lengths_from (*from_atom*: Union[int, libbase.UnifiedAtom], *dist_func*: molgraph.edgeweight_func_t (page 38)) → 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 (page 38)) → Optional[List[int]]

shortest_path_between (*from_atom*: int, *to_atom*: int) → Optional[List[int]]

shortest_path_between (*from_atom*: libbase.UnifiedAtom, *to_atom*: libbase.UnifiedAtom, *dist_func*: molgraph.edgeweight_func_t (page 38)) → Optional[List[int]]

shortest_path_between (*from_atom*: libbase.UnifiedAtom, *to_atom*: libbase.UnifiedAtom) → Optional[List[int]]

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 (page 38)) → float

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

shortest_path_length_between (*from_atom*: libbase.UnifiedAtom, *to_atom*: libbase.UnifiedAtom, *dist_func*: molgraph.edgeweight_func_t (page 38)) → float

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

Returns the length of the shortest path between two atoms.

4.1.7 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

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 as 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*)

A class representing the lattice of a `ChemicalSystem`.

cartesian_to_fractional (*cartesian_coord*: *numpy.ndarray*[*Any*, *numpy.dtype*[*numpy.float64*]])
 → *numpy.ndarray*[*Any*, *numpy.dtype*[*numpy.float64*]]
 Convert Cartesian coordinates to fractional coordinates for a single point.

cartesians_to_fractionals (*cartesian_coords*: *numpy.ndarray*[*Any*,
numpy.dtype[*numpy.float64*]]) → *numpy.ndarray*[*Any*,
numpy.dtype[*numpy.float64*]]
 Convert Cartesian coordinates to fractional coordinates for a set of points.

fractional_to_cartesian (*fractional_coord*: *numpy.ndarray*[*Any*, *numpy.dtype*[*numpy.float64*]])
 → *numpy.ndarray*[*Any*, *numpy.dtype*[*numpy.float64*]]
 Convert fractional coordinates to Cartesian coordinates for a single point.

fractionals_to_cartesians (*fractional_coords*: *numpy.ndarray*[*Any*,
numpy.dtype[*numpy.float64*]]) → *numpy.ndarray*[*Any*,
numpy.dtype[*numpy.float64*]]
 Convert fractional coordinates to Cartesian coordinates for a set of points.

get_angles () → *numpy.ndarray*[*Any*, *numpy.dtype*[*numpy.float64*]]
 Get the angles between the lattice vectors (note: always returns 3 numbers).

get_lengths () → *numpy.ndarray*[*Any*, *numpy.dtype*[*numpy.float64*]]
 Get the length of the lattice vectors (note: always returns 3 numbers).

get_vector_with_index (*i*: *int*) → *numpy.ndarray*[*Any*, *numpy.dtype*[*numpy.float64*]]
 Get the lattice vector of a given index.

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

is_orthogonal () → *bool*
 Is the lattice orthogonal?

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

property vectors
 The lattice vectors in bohr.

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_fractional_coordinates (*frac_coords*: *ArrayLike*) → *None*
 Sets the fractional coordinates of all atoms. In non-periodic directions the plain coordinates should be passed in bohr.

get_fractional_coordinates () → *numpy.ndarray*[*Any*, *numpy.dtype*[*numpy.float64*]]
 Returns the fractional coordinates of all atoms with respect to the cell. In non-periodic directions the plain coordinates are returned in bohr.

map_atoms (*start_range*: *Union*[*float*, *ArrayLike*]) → *Tuple*[*bool*, *numpy.ndarray*[*Any*,
numpy.dtype[*numpy.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 was applied to map the atoms into the cell.

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 convertible 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.

supercell (*supercell*: ArrayLike) → None

Create a supercell by scaling the lattice vectors.

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.

supercell_trafo (*supercell*: ArrayLike) → None

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

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.

density (*unit*: str = 'dalton/bohr3') → 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/bohr3') → None

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

4.1.8 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
```

(continues on next page)

(continued from previous page)

```

    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
"""
)

# 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 not 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-insensitive, so “A” and “a” correspond to the same region.

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

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

Returns a 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*) → numpy.ndarray[Any, numpy.dtype[numpy.int64]]

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

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

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

is_atom_in_region (*atom: Union[int, libbase.UnifiedAtom]*, *region: str*) → bool

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

is_atom_outside_region (*atom: Union[int, libbase.UnifiedAtom]*, *region: str*) → bool

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

get_regions_of_atom (*atom: Union[int, libbase.UnifiedAtom]*) → List[str]

Returns an alphabetically 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 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.

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.

add_atom_to_region (*atom: Union[int, libbase.UnifiedAtom]*, *region: str*) → None

Adds an atom to a region.

remove_atom_from_region (*atom: Union[int, libbase.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: Union[int, libbase.UnifiedAtom]*) → None

Removes an atom from all regions.

4.1.9 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: Union[int, libbase.UnifiedAtom]) → bool
        Checks if an atom is currently selected.

    get_selected_atoms () → numpy.ndarray[Any, numpy.dtype[numpy.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: Union[int, libbase.UnifiedAtom]) → None
        Selects an atom, i.e. adds it to the current selection.

    deselect_atom (atom: Union[int, libbase.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 = 'bohr'*) → 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[[libbase.UnifiedAtom], bool]*) → None

Selects atoms based on a predicate function.

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

Deselects atoms based on a predicate function.

select_connected_if (*pred: Callable[[libbase.UnifiedAtom, libbase.UnifiedAtom, libbase.UnifiedBond (page 37)], 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 if the predicate function returns `True`.

select_molecule_if (*pred: Callable[[libbase.UnifiedAtom, libbase.UnifiedAtom, libbase.UnifiedBond (page 37)], 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.

4.1.10 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
        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`

An optional list of all ADFProperties for all atoms.

property `forcefieldprops`

An optional list of all ForcefieldProperties for all atoms.

property `guiprops`

An optional list of all GUIProperties for all atoms.

enable_atomic_properties (*group_prefix*: Union[Literal[gui], Literal[adf], Literal[forcefield]])
 → None

Enables the use of a group of atomic properties.

disable_atomic_properties (*group_prefix*: Union[Literal[gui], Literal[adf], Literal[forcefield]])
 → None

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

atomic_properties_enabled (*group_prefix*: Union[Literal[gui], Literal[adf], Literal[forcefield]])
 → bool

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

has_ghost_atoms () → bool

Checks whether the ChemicalSystem contains any Ghost atoms.

Properties with in the atom class:

class UnifiedAtom

class UnifiedAtom (*Z*: int)

class UnifiedAtom (*element*: UnifiedElement)

class UnifiedAtom (*symbol*: T_AtomSymbol)

Class representing a single atom and its properties.

property mass

The mass of the atom in dalton.

property is_ghost

Whether the atom is a Ghost atom.

property adf

Atomic properties used by the ADF engine.

property forcefield

Atomic properties used by the AMS Forcefield engine.

property gui

Atomic properties used for visualization in the GUI.

Classes for the Engine/Module-Specific Properties:

class UnifiedGUIProperties

Atomic properties used for visualization in the GUI.

clear () → None

Unsets any GUI atomic properties.

property color

The color of an atom.

empty () → bool

Returns whether any of the GUI atomic properties is set.

property radius

The radius of an atom for visualization purposes.

class UnifiedADFPProperties

Atomic properties used by the ADF engine.

property ChgU

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

property EpsU

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

property SigU

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

clear () → None

Unsets any ADF atomic properties.

empty () → bool

Returns whether any of the ADF atomic properties is set.

property f

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

class UnifiedForcefieldProperties

Atomic properties used by the AMS Forcefield engine.

property charge

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

clear () → None

Unsets any Forcefield atomic properties.

empty () → bool

Returns whether any of the Forcefield atomic properties is set.

property type

The atom type to be used by the Forcefield engine.

4.1.11 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)
```

4.2 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: numpy.number) → numpy.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[numpy.number]) →
    List[numpy.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[numpy.number, ...]) → Tu-
    ple[numpy.number, ...]
classmethod convert (from_unit: str, to_unit: str, value: Tuple[Any, ...]) → Tuple[Any, ...]
classmethod convert (from_unit: str, to_unit: str, value: numpy.ndarray[Any,
    numpy.dtype[ScalarType]]) → numpy.ndarray[Any,
    numpy.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 51). Here is the output of `print(Units.units_summary_string())`:

```
=====
Quantity: ACCELERATION
=====

unit name                label
-----
angstrom/fs2             Å/fs²

=====
Quantity: ANGLE
=====
```

(continues on next page)

(continued from previous page)

| unit name | label |
|-------------------------|----------------------|
| ----- | ----- |
| degree | ° |
| rad | rad |
| grad | grad |
| turn | tr |
| ===== | |
| Quantity: CURRENT | |
| ===== | |
| unit name | label |
| ----- | ----- |
| ehartree/hbar | \hbar Ha/h |
| ampere | A |
| mampere | mA |
| uampere | μA |
| ===== | |
| Quantity: DENSITY | |
| ===== | |
| unit name | label |
| ----- | ----- |
| dalton/bohr3 | Da/Bohr ³ |
| kg/m3 | kg/m ³ |
| g/cm3 | g/cm ³ |
| kg1 | kg/L |
| ===== | |
| Quantity: DIPOLE | |
| ===== | |
| unit name | label |
| ----- | ----- |
| ebohr | \hbar Bohr |
| coulombmeter | Cm |
| debye | D |
| ===== | |
| Quantity: ELECTRICFIELD | |
| ===== | |
| unit name | label |
| ----- | ----- |
| hartree/ebohr | Ha/ \hbar Bohr |
| volt/angstrom | V/Å |
| volt/meter | V/m |

(continues on next page)

(continued from previous page)

```
=====
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 |
| 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 |

(continues on next page)

(continued from previous page)

```

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) |

```

=====
Quantity: MASS
=====
    
```

| unit name | label |
|-----------|----------------|
| dalton | Da |
| erestmass | m _e |
| proton | m _p |
| g | g |
| kg | kg |

```

=====
Quantity: MOMENTOFINERTIA
    
```

(continues on next page)

(continued from previous page)

```

=====
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
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

```

(continues on next page)

(continued from previous page)

```

=====
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
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

```

(continues on next page)

(continued from previous page)

| | |
|-----------------------|-----------------------|
| thz/bohr2 | THz/Bohr ² |
| ===== | |
| 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 |
| ----- | |
| bohr/fs | Bohr/fs |
| ===== | |
| Quantity: VOLUME | |
| ===== | |
| unit name | label |
| ----- | |
| bohr3 | Bohr ³ |
| angstrom3 | Å ³ |
| m3 | m ³ |

4.3 Input File

class **InputFile** (*program: str, text_input: str = "", string_leafs: 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*) → Union[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 58)

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, scm.plams.mol.molecule.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 58)
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 occurrences of a non-unique entry.

property program
The name of the program that this instance is reading input for.

read (*text_input: str, string_leafs: 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.

class InputError

Error thrown for issues with the input reading, e.g. reading non-existing keywords.

4.4 KFFile

See also:

The *AKFReader* (page 85) Python class provides a convenient alternative for reading data from KF files.

class KFFile (*filename: str, omode: OpenMode = <OpenMode.Any: 0>*)

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 60), to_kf: KFFile (page 60), section: str*) → None

copy_section (*from_kf: KFFile (page 60), to_kf: KFFile (page 60), 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 60)*) → 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.

property filename

The filename of the file managed by this KFFile instance.

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.

property path

The absolute path to the file managed by this KFFile instance.

read (*section: str = "", variable: str = "", return_as_list: bool = False, string_separator: Optional[StringSep (page 60)] = None*) → Union[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*) → numpy.ndarray[Any, numpy.dtype[numpy.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_logical_np (*section: str, variable: str*) → numpy.ndarray[Any, numpy.dtype[numpy.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*) → numpy.ndarray[Any, numpy.dtype[numpy.float64]]
 Reads real numbers from a specific section and variable into a numpy array.

read_section (*section: str*) → Dict[str, Union[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 = <StringSep.LCHARS: 2>*) → 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 60)
 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 60) = StringSep.LCHARS*) → None
Helper for @overload to raise when called.

class KFEError

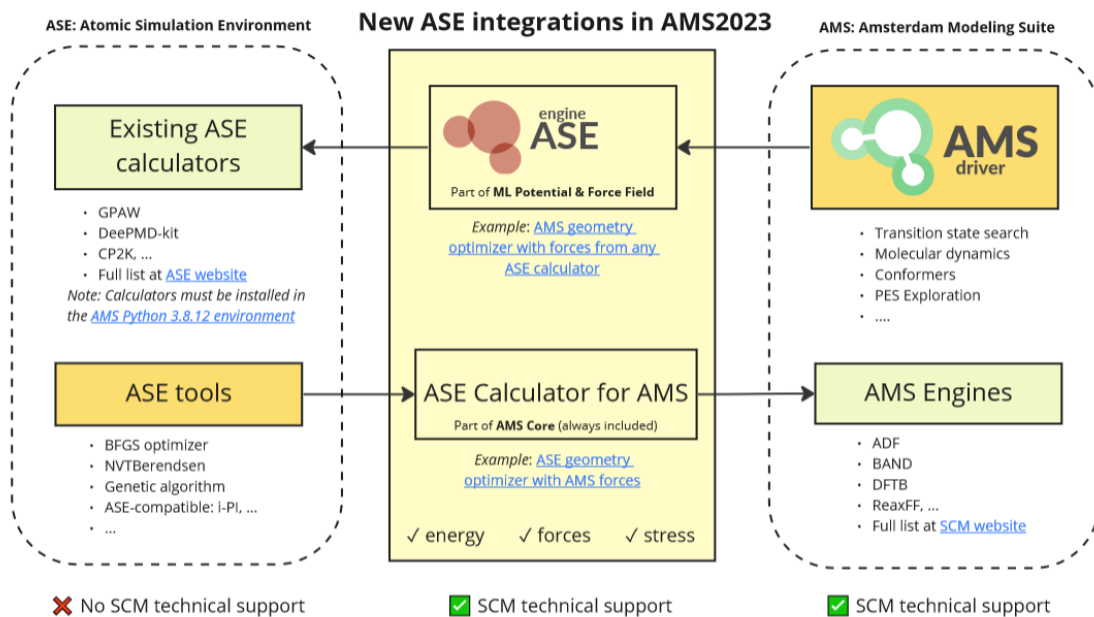
Error thrown for all KF issues, e.g. reading of non-existing variables or reading as wrong datatypes.

ASE

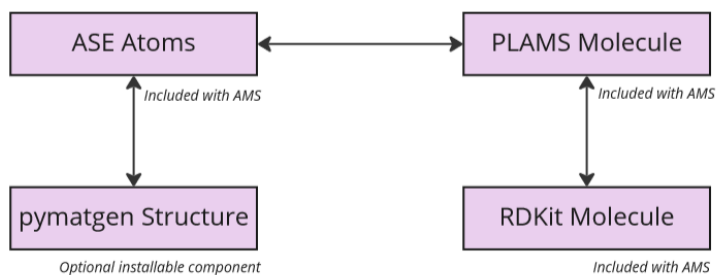
The [Atomic Simulation Environment \(ASE\)](https://wiki.fysik.dtu.dk/ase/) (<https://wiki.fysik.dtu.dk/ase/>) tool collection suite was designed as a flexible, easy-to-use, and customizable approach for the manipulation of quantum chemical models as well as for setting up and running the calculations required and for the analysis of the final results.

The development of ASE was originally started at the Technical University of Denmark but received also significant contributions from a large, international community. While ASE is [available](https://gitlab.com/ase/ase/tree/3.13.0/) (<https://gitlab.com/ase/ase/tree/3.13.0/>) under a GNU LGPL license, a modified version of this library is shipped together with the Amsterdam Modeling Suite.

PLAMS provides an interface to use [AMS as an ASE Calculator](#), while [Engine ASE](#) allows [ASE Calculators](#) (<https://wiki.fysik.dtu.dk/ase/ase/calculators/calculators.html#supported-calculators>) to be used within the AMS driver.



Chemical systems in Python



Note: On Windows machines the developers of the ASE library provide only a partial support, see [the ASE website](https://wiki.fysik.dtu.dk/ase/download.html#installation-on-windows) (<https://wiki.fysik.dtu.dk/ase/download.html#installation-on-windows>) for further details.

FLEXMD

FlexMD is a python library for Flexible multi-scale Molecular Dynamics simulation, developed by Rosa E. Buló, 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)* (page 65) [*I* (page 74)] 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 75)], 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/) (<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 74)] 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 74)], 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 74)]. 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:

$$EQM/MM(\text{Full}) = EMM(\text{Full}) + EQM(\text{QMRegion}) - EMM(\text{QMRegion})$$

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 **ForceJobs**, 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 Buló et al.[4 (page 74)] and P. Fleurat-Lessard et al.[8 (page 75)] 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 74)] 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 74)] and umbrella sampling [6 (page 75)]. 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 75)]. 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 69) and *Molecular Dynamics* (page 69) before working with FlexMD. Basic understanding of the [Python Programming Language](http://python.org/) (<http://python.org/>) is also required. The Python website hosts documentation and a [tutorial](http://docs.python.org/2/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 through 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 74)] 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 its 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 MDPPropagator 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. Buló, *A python library for flexible multi-scale molecular dynamics simulations*. <http://www.scm.com>

6.8.1 External programs and Libraries

Click [here](#) for the list of programs and/or libraries used in the ADF package. On some platforms optimized libraries have been used and/or vendor specific MPI implementations.

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. Buló, 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. Buló, 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\(95\)00053-1](https://doi.org/10.1016/0010-4655(95)00053-1))
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>)

PLAMS

PLAMS (Python Library for Automating Molecular Simulation) is a collection of tools that aim at providing powerful, flexible and easily extendable Python interface to molecular modeling programs. It takes care of input preparation, job execution, file management and output processing as well as helps with building more advanced data workflows.

- [PLAMS tutorials](#)
- [PLAMS documentation](#)

7.1 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 PLAMS: PLAMS, written by Michał Handzlik, <http://www.scm.com>

7.1.1 External programs and Libraries

[Click here](#) for the list of programs and/or libraries used in the ADF package. On some platforms optimized libraries have been used and/or vendor specific MPI implementations.

AUTOGRAFS

The [Automatic Topological Generator for Framework Structures](http://pubs.acs.org/doi/abs/10.1021/jp507643v) (<http://pubs.acs.org/doi/abs/10.1021/jp507643v>) is a software for the design and 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 *ASE* (page 65) 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) (<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.

8.1 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/) (<http://rcsr.anu.edu.au/>) and [EPINET](http://epinet.anu.edu.au/) (<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.

8.1.1 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",
               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 hydrogen 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.

8.1.2 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.
↳5:4/1.5:10/1.0

```

(continues on next page)

(continued from previous page)

| | | | | | |
|-----------------|-------------|-------------|-------------|------------|--------------------|
| 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 | | | | | |

8.1.3 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")

```

8.2 AuToGraFS Examples

the following examples work with all calculators of the Amsterdam Modeling Suite. Any python script using these libraries should be launched using the “\$AMSBIN/amspython” binary.

8.2.1 Generation of all available pillared SURMOF

```
# Required imports
from scm.autografs import *
from itertools import combinations_with_replacement

# create the generator object
mofgen = autografs.Autografs(path="./my_database/")

# the SURMOF family of frameworks is
topology = "pcu"

# get all available linkers, center, pillars for the pcu topology
centers = mofgen.get_available_centers(topology=topology)
for center in centers:
    # the pcu topology accepts up to three different linkers.
    # here, we'll use two and a pillar
    linkers = mofgen.get_available_linkers(topology=topology, center=center)
    for linker in combinations_with_replacement(linkers, 2):
        pillars = mofgen.get_available_pillars(topology=topology, center=center)
        for pillar in pillars:
            label = "{0}-{1}-{2}".format(center, linker,pillar)
            # generate this particular framework
            framework = mofgen.make(label=label,
                                   topology=topology,
                                   center=center,
                                   linker=linker,
                                   pillar=pillar)

            # write the UFF inpufile under "label.run" and "label.ams"
            framework.write()
```

8.2.2 Generation of a defectuous UIO-66 MOF from custom files

```
# Required imports
from scm.autografs import *
from random import gauss

# get the building units as Fragment objects
center = utils.read_inp("UIO66_center.inp")
linker = utils.read_inp("UIO66_linker.inp")

# instantiate the generator
mofgen = autografs.Autografs()

# generate the MOF
mof = mofgen.make(label="UIO66", topology="bcu", center=center, linker=linker)

# get the 3*3*3 supercell
```

(continues on next page)

(continued from previous page)

```
supercell = mof * (3,3,3)

# introduce 10 defects in a gaussian distribution around the center of the
↳supercell
# the connections are capped with hydrogen.
indices = []
mu      = len(supercell)/2
sigma  = len(supercell)/4
number_of_defects = 10
while len(indices) < number_of_defects:
    index = int(abs(gauss(mu, sigma)))
    if (index not in indices) and index < len(supercell):
        indices.append(index)
supercell.insert_defect(indices = indices)

# view resulting framework in adfinput
supercell.view()
```

8.2.3 Generation of conformers in the IRMOF-5

```
# Required imports
from scm.autografs import *
from itertools import combinations_with_replacement

# instantiate the generator
mofgen = autografs.Autografs()

# generate the MOF
mof = mofgen.make(label="IRMOF-5", topology="pcu", center="mof5", linker="benzene")

# get the indices of the benzenes in the mof
benzenes = mof.get_linkers()
choice_angles = [0.0, 45.0, 90.0]

# generate all possible combinations
iteration = 0
for angles in combinations_with_replacement(choice_angles, 3):
    conformer = mof.copy()
    for benzene, angle in zip(benzenes, angles):
        conformer.rotate_fragment(index=benzene, angle=angle)
    conformer.write(name="conformer-{}".format(iteration))
    iteration += 1
```


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

```
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")
print(f"Energy: {energy_in_eV} [eV]")
```

(continues on next page)

(continued from previous page)

```
# 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 *KFFFile* (page 60) Python class can also be used to read KF files. Key differences between the *AKFReader* and the *KFFFile*:

- 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 *KFFFile* 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 20)) 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(1)’ -> [(‘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”.

Non-alphabetical

`__add__()` (*UnifiedChemicalSystem method*), 32
`__iadd__()` (*UnifiedChemicalSystem method*), 32

A

`add_atom()` (*UnifiedChemicalSystem method*), 31
`add_atom_to_region()` (*UnifiedChemicalSystem method*), 44
`add_atoms_to_region()` (*UnifiedChemicalSystem method*), 44
`add_bond()` (*UnifiedBonds method*), 37
`add_bonds()` (*UnifiedBonds method*), 37
`add_other()` (*UnifiedChemicalSystem method*), 32
`adf()` (*UnifiedAtom property*), 48
`adfprops()` (*UnifiedChemicalSystem property*), 47
AKFReader (*class in scm.akfreader*), 86
`align_to()` (*UnifiedChemicalSystem method*), 35
amsprep module, 3
amsreport module, 11
`apply_strain()` (*UnifiedChemicalSystem method*), 41
`apply_strain_voigt()` (*UnifiedChemicalSystem method*), 42
`atom_index()` (*UnifiedChemicalSystem method*), 31
`atom_is_in_ring()` (*UnifiedChemicalSystem method*), 38
`atomic_properties_enabled()` (*UnifiedChemicalSystem method*), 48
`atoms()` (*UnifiedChemicalSystem property*), 31
`atoms_are_bonded()` (*UnifiedBonds method*), 37

B

`bond_cuts_molecule()` (*UnifiedChemicalSystem method*), 38
`bonds()` (*UnifiedChemicalSystem property*), 37

C

`cartesian_to_fractional()` (*UnifiedLattice method*), 40
`cartesians_to_fractionals()` (*UnifiedLattice method*), 41
`center_of_mass()` (*UnifiedChemicalSystem method*), 35

`charge()` (*UnifiedChemicalSystem property*), 33
`charge()` (*UnifiedForcefieldProperties property*), 49
`ChgU()` (*UnifiedADFPProperties property*), 48
`clear()` (*UnifiedADFPProperties method*), 49
`clear()` (*UnifiedForcefieldProperties method*), 49
`clear()` (*UnifiedGUIProperties method*), 48
`close()` (*KFFFile method*), 60
`close_var()` (*KFFFile method*), 60
`color()` (*UnifiedGUIProperties property*), 48
`contains_atom()` (*UnifiedChemicalSystem method*), 31
`convert()` (*Units class method*), 50
`coords()` (*UnifiedAtom property*), 33
`coords()` (*UnifiedChemicalSystem property*), 31
`copy_section()` (*KFFFile method*), 60
cpkf module, 18
`create_section()` (*KFFFile method*), 60
`create_var()` (*KFFFile method*), 60

D

`delete_file()` (*KFFFile class method*), 60
`delete_section()` (*KFFFile method*), 61
`delete_var()` (*KFFFile method*), 61
`density()` (*UnifiedChemicalSystem method*), 42
`description()` (*AKFReader method*), 86
`deselect_all()` (*UnifiedChemicalSystem method*), 46
`deselect_atom()` (*UnifiedChemicalSystem method*), 45
`deselect_atoms()` (*UnifiedChemicalSystem method*), 45
`deselect_atoms_if()` (*UnifiedChemicalSystem method*), 46
`disable_atomic_properties()` (*UnifiedChemicalSystem method*), 48
dmpkf module, 18
`do_regions_intersect()` (*UnifiedChemicalSystem method*), 43

E

`empty()` (*UnifiedADFPProperties method*), 49
`empty()` (*UnifiedForcefieldProperties method*), 49
`empty()` (*UnifiedGUIProperties method*), 48

- enable_atomic_properties() (*UnifiedChemicalSystem method*), 47
- EpsU() (*UnifiedADFPProperties property*), 48
- extract_atoms() (*UnifiedChemicalSystem method*), 33
- ## F
- f() (*UnifiedADFPProperties property*), 49
- filename() (*KFFFile property*), 61
- find() (*AKFReader method*), 86
- forcefield() (*UnifiedAtom property*), 48
- forcefieldprops() (*UnifiedChemicalSystem property*), 47
- formula() (*UnifiedChemicalSystem method*), 33
- fractional_to_cartesian() (*UnifiedLattice method*), 41
- fractionals_to_cartesians() (*UnifiedLattice method*), 41
- from_in() (*UnifiedChemicalSystem class method*), 30
- from_input() (*UnifiedChemicalSystem class method*), 30
- from_kf() (*UnifiedChemicalSystem class method*), 30
- from_xyz() (*UnifiedChemicalSystem class method*), 30
- ## G
- geometric_center() (*UnifiedChemicalSystem method*), 35
- get() (*InputFile method*), 58
- get_angles() (*UnifiedLattice method*), 41
- get_atoms_in_region() (*UnifiedChemicalSystem method*), 44
- get_atoms_outside_region() (*UnifiedChemicalSystem method*), 44
- get_bonds_for_atom() (*UnifiedBonds method*), 37
- get_category() (*InputFile method*), 58
- get_entries_in_order() (*InputFile method*), 58
- get_factor() (*Units class method*), 50
- get_fractional_coordinates() (*UnifiedChemicalSystem method*), 41
- get_freeblock() (*InputFile method*), 58
- get_header() (*InputFile method*), 58
- get_int() (*InputFile method*), 58
- get_int_list() (*InputFile method*), 58
- get_json() (*InputFile method*), 58
- get_lengths() (*UnifiedLattice method*), 41
- get_logical() (*InputFile method*), 58
- get_quantities() (*Units class method*), 51
- get_real() (*InputFile method*), 59
- get_real_list() (*InputFile method*), 59
- get_regions_of_atom() (*UnifiedChemicalSystem method*), 44
- get_selected_atoms() (*UnifiedChemicalSystem method*), 45
- get_skeleton() (*KFFFile method*), 61
- get_string() (*InputFile method*), 59
- get_system_blocks_as_mols() (*InputFile method*), 59
- get_type() (*InputFile method*), 59
- get_unit() (*Units class method*), 51
- get_units() (*Units class method*), 51
- get_vector_with_index() (*UnifiedLattice method*), 41
- get_volume() (*UnifiedLattice method*), 41
- guess_bonds() (*UnifiedChemicalSystem method*), 38
- gui() (*UnifiedAtom property*), 48
- guiprops() (*UnifiedChemicalSystem property*), 47
- ## H
- has_ghost_atoms() (*UnifiedChemicalSystem method*), 48
- has_lattice() (*UnifiedChemicalSystem method*), 40
- has_region() (*UnifiedChemicalSystem method*), 43
- ## I
- in_chemicalsystem() (*UnifiedAtom method*), 33
- inertia_tensor() (*UnifiedChemicalSystem method*), 35
- InputError (*class in libbase*), 59
- InputFile (*class in libbase*), 58
- InputFile.EntryCategory (*class in libbase*), 58
- InputFile.EntryType (*class in libbase*), 58
- invert_selection() (*UnifiedChemicalSystem method*), 46
- is_atom_in_region() (*UnifiedChemicalSystem method*), 44
- is_atom_outside_region() (*UnifiedChemicalSystem method*), 44
- is_atom_selected() (*UnifiedChemicalSystem method*), 45
- is_defined() (*InputFile method*), 59
- is_ghost() (*UnifiedAtom property*), 48
- is_kffile() (*KFFFile class method*), 61
- is_linear() (*UnifiedChemicalSystem method*), 35
- is_ordered() (*InputFile method*), 59
- is_orthogonal() (*UnifiedLattice method*), 41
- is_present() (*InputFile method*), 59
- is_unique() (*InputFile method*), 59
- is_valid_region_name() (*UnifiedChemicalSystem class method*), 43
- ## K
- KF command line utilities, 18
- KFError (*class in libbase*), 63
- KFFFile (*class in libbase*), 60
- KFFFile.OpenMode (*class in libbase*), 60
- KFFFile.StringSep (*class in libbase*), 60
- KFFFile.VarType (*class in libbase*), 60

L

`lattice()` (*UnifiedChemicalSystem* property), 40
`lattice_displacements()` (*UnifiedBond* property), 37

M

`make_selection_cappable()` (*UnifiedChemicalSystem* method), 46
`map_atoms()` (*UnifiedChemicalSystem* method), 41
`mass()` (*UnifiedAtom* property), 48
`molecule_indices()` (*UnifiedChemicalSystem* method), 38
`molgraph_dijkstra()` (*UnifiedChemicalSystem* method), 38
`molgraph_edgweight_func_t()` (*UnifiedChemicalSystem* property), 38
`moments_of_inertia()` (*UnifiedChemicalSystem* method), 35

N

`num_atoms_in_region()` (*UnifiedChemicalSystem* method), 43
`num_atoms_outside_region()` (*UnifiedChemicalSystem* method), 43
`num_bonds()` (*UnifiedBonds* method), 37
`num_molecules()` (*UnifiedChemicalSystem* method), 38
`num_regions()` (*UnifiedChemicalSystem* property), 43
`num_sections()` (*KFFFile* method), 61
`num_selected_atoms()` (*UnifiedChemicalSystem* method), 45
`num_variables()` (*KFFFile* method), 61
`num_vectors()` (*UnifiedLattice* property), 41
`number_of_entries()` (*InputFile* method), 59

O

`open_var()` (*KFFFile* method), 61
`order()` (*UnifiedBond* property), 37

P

`path()` (*KFFFile* property), 61
`perturb_coordinates()` (*UnifiedChemicalSystem* method), 35
`perturb_lattice()` (*UnifiedChemicalSystem* method), 42
pkf module, 18
`print_all_variables()` (*AKFReader* method), 86
`print_variable()` (*AKFReader* method), 86
`program()` (*InputFile* property), 59

R

`radius()` (*UnifiedGUIProperties* property), 48
`read()` (*AKFReader* method), 86

`read()` (*InputFile* method), 59
`read()` (*KFFFile* method), 61
`read_int()` (*KFFFile* method), 61
`read_ints()` (*KFFFile* method), 61
`read_ints_np()` (*KFFFile* method), 61
`read_logical()` (*KFFFile* method), 61
`read_logicals()` (*KFFFile* method), 61
`read_logicals_np()` (*KFFFile* method), 62
`read_real()` (*KFFFile* method), 62
`read_reals()` (*KFFFile* method), 62
`read_reals_np()` (*KFFFile* method), 62
`read_section()` (*KFFFile* method), 62
`read_string()` (*KFFFile* method), 62
`read_strings()` (*KFFFile* method), 62
`region_names()` (*UnifiedChemicalSystem* property), 43
`remove_atom()` (*UnifiedChemicalSystem* method), 32
`remove_atom_from_all_regions()` (*UnifiedChemicalSystem* method), 44
`remove_atom_from_region()` (*UnifiedChemicalSystem* method), 44
`remove_atoms()` (*UnifiedChemicalSystem* method), 32
`remove_atoms_from_region()` (*UnifiedChemicalSystem* method), 44
`remove_bond()` (*UnifiedBonds* method), 37
`remove_bonds_between_atoms()` (*UnifiedBonds* method), 37
`remove_bonds_to_atom()` (*UnifiedBonds* method), 37
`remove_bonds_to_atoms()` (*UnifiedBonds* method), 37
`remove_region()` (*UnifiedChemicalSystem* method), 44
`rewind_var()` (*KFFFile* method), 62
`rmsd()` (*UnifiedChemicalSystem* class method), 35
`rotate()` (*UnifiedChemicalSystem* method), 35
`rotation_matrix_minimizing_rmsd()` (*UnifiedChemicalSystem* class method), 35

S

`save()` (*KFFFile* method), 62
`section_exists()` (*KFFFile* method), 62
`sections()` (*KFFFile* method), 62
`select_all()` (*UnifiedChemicalSystem* method), 46
`select_atom()` (*UnifiedChemicalSystem* method), 45
`select_atom_close_to_origin()` (*UnifiedChemicalSystem* method), 46
`select_atoms()` (*UnifiedChemicalSystem* method), 45
`select_atoms_if()` (*UnifiedChemicalSystem* method), 46
`select_atoms_of_same_type()` (*UnifiedChemicalSystem* method), 46
`select_connected()` (*UnifiedChemicalSystem* method), 46

- select_connected_if() (*UnifiedChemicalSystem method*), 46
 select_molecule() (*UnifiedChemicalSystem method*), 46
 select_molecule_if() (*UnifiedChemicalSystem method*), 46
 select_region() (*UnifiedChemicalSystem method*), 46
 select_within_radius() (*UnifiedChemicalSystem method*), 46
 set_atom() (*UnifiedChemicalSystem method*), 32
 set_atoms_in_region() (*UnifiedChemicalSystem method*), 44
 set_density() (*UnifiedChemicalSystem method*), 42
 set_fractional_coordinates() (*UnifiedChemicalSystem method*), 41
 set_selected_atoms() (*UnifiedChemicalSystem method*), 45
 shortest_path_between() (*UnifiedChemicalSystem method*), 39
 shortest_path_length_between() (*UnifiedChemicalSystem method*), 39
 shortest_path_lengths_from() (*UnifiedChemicalSystem method*), 39
 SigU() (*UnifiedADFPProperties property*), 48
 skip_var() (*KFFFile method*), 62
 split() (*UnifiedChemicalSystem method*), 33
 split_into_molecules() (*UnifiedChemicalSystem method*), 38
 supercell() (*UnifiedChemicalSystem method*), 42
 supercell_trafo() (*UnifiedChemicalSystem method*), 42
 symbol() (*UnifiedAtom property*), 33
- ## T
- total_mass() (*UnifiedChemicalSystem method*), 33
 translate() (*UnifiedChemicalSystem method*), 34
 type() (*UnifiedForcefieldProperties property*), 49
- ## U
- udmpkf module, 18
 UnifiedADFPProperties (*class in libbase*), 48
 UnifiedBond (*class in libbase*), 37
 UnifiedForcefieldProperties (*class in libbase*), 49
 UnifiedGUIProperties (*class in libbase*), 48
 UnifiedLattice (*class in libbase*), 40
 Units (*class in libbase*), 50
 units_summary_string() (*Units class method*), 51
 UnitsError (*class in libbase*), 51
- ## V
- var_exists() (*KFFFile method*), 62
 var_length() (*KFFFile method*), 62
- ## W
- write() (*KFFFile method*), 62
 write_in() (*UnifiedChemicalSystem method*), 30
 write_kf() (*UnifiedChemicalSystem method*), 30
 write_xyz() (*UnifiedChemicalSystem method*), 31
- ## Z
- Z() (*UnifiedAtom property*), 33