Scripting Manual

ADF Modeling Suite 2016

www.scm.com

November 18, 2016

CONTENTS

CHAPTER

ONE

COMMAND LINE TOOLS

- **adfprep**: prepare an ADF job from a script (or command line).
- **adfreport**: 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.

1.1 ADFprep: generate (multiple) ADF jobs

ADFprep allows one to generate input files for the different programs of the ADF modeling suite by means of console commands. As such ADFprep 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.

ADFprepare ($ADFBIN/adfprep) generates a job script from a template .adf file. Such a template file can either be produced by ADFinput or simply be found among the default templates included. These default templates are identical to those present in ADFinput.

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

- In BakersetSP you will see how to use adfprep 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. ADFreport is used to collect the values of the bonding energies resulting from these calculations.
- In ConvergenceTestCH4 you will see how to use ADFprep to test convergence of the bonding energy with respect to the basis set and the numerical integration grid.

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

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

Usage: adfprep -t template.adf [-m molecule.(adf|xyz|mol|t21)] [-z charge] [-s spin]
                               [-runtype SinglePoint|GeometryOptimization|Frequencies]
                               [-gradientsonly]
                               [-q quality] [-zlmfit quality] [-kspace quality]
                               [-lattice v1.x v1.y v1.z ...]
                               [-i integration] [-b basis] [-c core] [-r relativity]
                               [-basiscacheid id]
                               [-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 adffile]
                                [-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 ADFinput, also for menu choices.

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

    Instead of a .adf file, you may also specify the name of one of the standard templates
    as defined in ADFinput: "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 ADFinput can import,
    for example .adf, .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 .adf 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

    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:
        $ADFHOME/atomicdata/Molecules/Crystals/Cubic/CsCl.cry#MgTl

    When looping, all resulting jobs will be joined together, the jobname and adf 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 ADFinput
       atom is the selected atom, structure is the name of the structure file
       the arguments must be enclosed in quotes, and may be repeated for multiple changes
-liststructures: list available structure files for use with -structure, and exit

-runtype: run type (SinglePoint,GeometryOptimization,Frequencies)
-gradientsonly: after calculating the gradients, stop
                works also for excited state gradients if requested in your template
-z: charge (real number)
-s: spin (integer), if not zero this implies an unrestricted calculation
-q: quality (Basic, Normal, Good, VeryGood or Excellent), default for Becke/ZlmFit
-i: integration (integer)
-i: Becke integration (Basic, Normal, Good, VeryGood or Excellent)
-i: teVelde integration (integer)
-zlmfit: ZlmFit quality (Basic, Normal, Good, VeryGood or Excellent)
-kspace: KSpace quality (GammaOnly, Basic, Normal, Good, VeryGood or Excellent)
-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 ADFinput:
        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 ADFinput for some option, see bin/adfinput.tcl/tpl/Defaults.tpl
         value: set gin(key) to the specified value
 -nochain: unset chain option (used internally by chain jobs)

OUTPUT
-bondsonly: only the bonds as generated by the GUI will be exported (the GUIBONDS block)
-logfile: force the specified logfile to be used in the run script
-j: produce a fully runnable job (as the .job files from ADFjobs),
    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 ADFinput, files are left as they are)
-a: save a .adf file that matches the run script, except for the -k arguments
    (they are listed in the user input field)
    adffile is the name of the adffile, including the .adf extension (required)

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

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

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

1.1.1 Additional Notes

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

1.2 ADFreport: generate reports

The utility ADFreport ($ADFBIN/adfreport) 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 ADFreport 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.

```
adfreport job.t21 BondingEnergy
```

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

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

The options of ADFreport 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.

```
adfreport -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 .adf 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 ADFreport will be stored. The output will be printed to standard output if this option is absent.

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

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

-v command line to pass to adfview (without filenames) to generate images. The image will be generated by ADFview 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 ADFreport 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 ADF modeling suite.

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

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

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

`title` title of the calculation

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

`weight` molecular weight

`symmetry` molecular symmetry

`natoms` number of atoms

`integration` integration accuracy

`integration-min` minimum integration accuracy

`integration-max` maximum integration accuracy

scfstatus SCF convergence status

charge the requested charge

charges shorthand for Voronoi, Hirshfeld and Mulliken charges

voronoi Voronoi deformation charges

hirshfeld Hirshfeld fragment charges, atomic fragment definition required

mdc All available MDC atom charges

mdc-m MDC-M charges

mdc-d MDC-D charges

mdc-q MDC-Q charges

mulliken Mulliken charges

bondorders Mayer bond orders

nmr NMR shieldings

nmr-shieldings NMR shieldings

nmr-shielding-tensor NMR shielding tensor

nmr-j-coupling-tensor NMR j coupling tensor

nmr-k-coupling-tensor NMR k coupling tensor

nmr-j-coupling-constant NMR j coupling constant

nmr-k-coupling-constant NMR k coupling constant

dipolev* dipole vector

dipole dipole moment (length of dipole vector)

quadrupole quadrupole tensor

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

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

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

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

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

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

atomlabels name of atoms with sequence number, starting at 0

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

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

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

spin the requested spin polarization

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

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

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

sdf geometry in SDF format

bgf geometry in BGF format

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

labels (optional): include atom labels in output

format (optional): format field

atom numbers, starting at 1, in input order

examples

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

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

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

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

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

energies* all avaliable 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 interation) 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

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

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

`pdbtrajectory-(nobonds|usepdbinfo)`

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

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

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

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

   m : print list of molecule names and formulas only

   x : allow xyz only frames (missing velocities)

   f : add forces if available

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

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

Special features for ReaxFF parameter optimization: a geo file in biograph format can be converted fr
```

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 avaliable energies (bonding up to xc, with labels), fmt option after #

1.2.1 Additional notes

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

1.3 KF command line utilities

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

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

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

pkf

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

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

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

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

```
pkf file > ascii_result
```

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

The output of pkf consists of:

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

cpkf

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

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

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

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

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

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

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

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

dmpkf

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

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

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

The format to be used for the individual keys:

```
Sec%Var
```

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

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

```
dmpkf file > ascii_result
```

The output contains for each printed variable:

- One line with the name of the section it belongs to;
- One line with the name of the variable itself;
- One line with three integers:
  - The amount of space reserved for the variable on the file which is, however, relevant for programs operating with KF files only;
  - The amount of data associated with the variable: for reals, integers, logicals: the number of such elements; for strings: the number of characters;
  - An integer code for the data type of the variable: 1=integer, 2=real, 3=character, 4=logical;
- The values of the variable (on as many lines as necessary): for scalar variables only one value, for arrays as many values as the array contains.

udmpkf

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

```
udmpkf file
```

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

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

```
dmpkf TAPE21 > t21_ASCII
```

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

```
udmpkf < t21_ASCII TAPE21_new
```

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

CHAPTER

TWO

ASE

The Atomic Simulation Environment (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://wiki.fysik.dtu.dk/ase/download.html#download) under a GNU LGPL license, a modified version of this library is shipped together with the ADF modeling suite.

This latter version of ASE provides the interfaces to ADF, BAND, DFTB, and ReaxFF. In addition, the shipped version was extended by several command line scripts that allow one to perform tasks such as geometry and transition state searches in terms of single-line commands.

The interfaces of ASE to the ADF modeling suite were written by Damien Coupry and Thomas M. Soini.

While the reader is encouraged to consult the very detailed ASE manual (https://wiki.fysik.dtu.dk/ase/ase/ase.html) for the clarification of technical details, an overview about the general concepts and mechanisms behind ASE is provided. The rest of this section is then dedicated to a documentation and a demonstration of the usage of the extensions to ASE provided in the ADF modeling suite.

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) for further details.

2.1 General Concepts

The ASE library makes extensive use of the object oriented programming features included in Python. Objects (https://en.wikipedia.org/wiki/Object_(computer_science)) and their corresponding classes (https://en.wikipedia.org/wiki/Class_(computer_programming)) are therefore the most relevant entities in ASE-based scripts. While the ASE library consists of very many different classes, they can be subdivided into the following three main groups described in the following section.

2.1.1 Components of ASE

Quantum Chemical Model

Objects like those defined by the Atoms class (https://wiki.fysik.dtu.dk/ase/ase/atoms.html#module-ase.atoms) represent the core part of ASE. Different ways to construct Atoms objects are provided by the ASE library, including constructors for specific types of quantum chemical systems such as bulk materials and surfaces, clusters, nanotubes etc. During runtime an Atoms object always comprises all available information about the quantum chemical

model it describes. Furthermore, the Atoms class includes the essential methods for manipulating the atomic model (https://wiki.fysik.dtu.dk/ase/ase/atoms.html#ase.atoms.Atoms) under study.

While Atoms (https://wiki.fysik.dtu.dk/ase/ase/atoms.html#module-ase.atoms) is by far most frequently encountered class in ASE scripts, several other objects have similar purposes in more specialized respective contexts. The Constraints and Filter classes (https://wiki.fysik.dtu.dk/ase/ase/constraints.html#module-ase.constraints) are important examples for such cases and serve e.g. to enforce user defined constraints on the system upon geometry relaxation.

Quantum Chemical Calculation

Calculator classes (https://wiki.fysik.dtu.dk/ase/ase/calculators/calculators.html#module-ase.calculators) provide the definition of the quantum chemical computation required to obtain the desired physical property of models defined by the Atoms objects. Such classes essentially interface ASE to a specific quantum chemistry program and contain methods necessary to transfer input information to this program as well as to retrieve the corresponding results after the completion of the program run.

Calculators are essential for the aforementioned Atoms object to be fully applicable. As an example, most physical properties of a system represented by an Atoms object require a prescription about how to actually compute the result, which is in turn mediated by the assigned Calculator class. Due to that, Calculator objects usually become part of the Atoms object during an ASE run (see also the next section).

The ASE variant within the ADF modeling suite includes four additional Calculator classes which enable computations with ADF, BAND, DFTB, and ReaxFF from within ASE, respectively.

Simulation Definitions

Finally, a third type of class (denoted as Simulation object in the following) represents the actual simulation which is conducted for a given combination of Atoms and Calculator objects. Such a Simulation object can for example either represent a simple geometry relaxation (https://wiki.fysik.dtu.dk/ase/ase/optimize.html#module-ase.optimize), a (numerical) frequency (https://wiki.fysik.dtu.dk/ase/ase/vibrations.html#module-ase.vibrations) or phonon calculation (https://wiki.fysik.dtu.dk/ase/ase/phonons.html#module-ase.phonons), a transition state search (https://wiki.fysik.dtu.dk/ase/ase/neb.html?highlight=transition%20state), or an entire gobal optimization (https://wiki.fysik.dtu.dk/ase/ase/ga.html#module-ase.ga) or molecular dynamics simulation (https://wiki.fysik.dtu.dk/ase/ase/md.html#module-ase.md).

2.1.2 A Typical ASE Run

In a standard ASE run objects of the three kinds presented above interact in order to perform the intended calculations. A typical ASE run is organized as follows

Definition and Initialization of Objects Mind that there are alternative, usually more convenient ways to define a chemical system from file inputs or system specific presets

```
MySystem     = Atoms( <initializing variables> )
MyCalculator = <CalculatorClass>( <options> )
```

Calculator → Atoms The Calculator object is assigned to the Atoms object and becomes part of it.

```
MySystem.set_calculator( MyCalculator )
```

The methods of the Atoms object that refer to calculated results can then directly call methods from the Calculator.

Atoms → Simulation The Atoms object is used as one of the initialization variables for the constructor of the Simulation object.

```
MySimulation = <SimulatorClass>( MySystem, <options> )
```

The Simulation object is now able to automatically alter the System object (e.g. by setting new atomic coordinates during a geometry relaxation) as well as to request the calculation of the required physical properties for the current status of the system. The simulation can be started as follows.

```
MySimulation.run( <options> )
```

Evaluation It is sometimes convenient to print or further evaluate the result e.g. by calculating the reaction energies between two different Systems

```
MyReactands = Atoms(..., calculator = MyCalculator(), ...)
MyProducts  = ...

# Relax both, MyReactands and MyProducts
...

E_R = MyReactands.get_potential_energy()
E_P = MyProducts.get_potential_energy()

print 'ReactionEnergy (eV) = ', E_P - E_R
```

Mind that ASE exclusively uses eV and Å as units of energy and length, respectively.

2.2 SCM Calculators in ASE

ASE calculators are implemented for ADF, BAND, DFTB, and ReaxFF. These will behave almost exactly like native ASE calculators, with the notable exception that they will by default use the SCM drivers for geometry optimization, since they perform significantly better both in convergence test and number of steps to convergence.

2.2.1 ADF Calculator

the ADF class

Class for SCMSUITE ADF ASE calculator. It is imported and used the following way: eg:

```
from ase.calculators.scm import ADF

some_molecule = ase.Atoms(...)
some_molecule.set_calculator(ADF(label=some_mol))
```

Will create a directory "some_mol", in which some_mol.run will be found and executed. Note that it requires at least a label as argument, and that the Atoms object has to be non periodic.

Arguments and default values

The following options can be passed to the constructor:

template: string, name of a .adf file containing the calculation details. Any other argument passed to the calculator will override this template.

restart: string, prefix for a restart file. May contain a directory. Default is None, so the calculations don't restart except if specified.

ignore_bad_restart_file: True or False, Ignore broken or missing restart file. By default, the scripts throw errors if the restart file is missing or broken.

label: string, base name used for all created files. May contain a directory.

atoms: ASE Atoms object, Optional Atoms object to which the calculator will be attached. When restarting, atoms will get its positions and unit-cell updated from file.

job: type of job this calculator will run, Defaults to GeometryOptimization, in which case any method like "get_potential_energy()" used on the attached atoms object will use the native ADF drivers to perform a geometry optimization. To use an ASE optimizer like BFGS or FIRE, set this option to SinglePoint.

- `SinglePoint`
- `GeometryOptimization`
- `Frequencies`
- `None, only use if template is given`

Calculation arguments: these will be used for the ADF job construction.

- `charge = Charge` of the Atoms object, defaults to `0`
- `spin = Spin` of the Atoms object, defaults to `0`
- `quality = Quality` of the models, defaults to `Normal`:
  - `Basic`
  - `Normal`
  - `Good`
  - `VeryGood`
  - `Excellent`
- `integration = Integration` quality , defaults to `Normal`:
  - for Becke scheme:
    - `Basic`
    - `Normal`
    - `Good`
    - `VeryGood`
    - `Excellent`
  - for TeVelde scheme:
    - `3`
    - `4`
    - `5`
    - `6`
    - `10`
- `basis = Basis` set, defaults to DZ:
  - `SZ`
  - `DZ`

  - DZP
  - TZP
  - TZ2P
  - QZ4P
- `frozencore = Size` of the Frozen Core, defaults to `Large`:
  - None
  - Small
  - Medium
  - Large
- `relativity = Level` of relativity theory to use, defaults to `None`:
  - None
  - Scalar
  - Spin-Orbit
- `xc = Exchange-Corellation potential` to use during SCF, defaults to `LDA`:
  - 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
- `postxc = Exchange-Corellation energy` to use after SCF, defaults to `Default`:
  - Default
  - LDA+GGA_METAGGA
  - LDA+GGA+METAGGA+HYBRIDS

2.2.2 BAND Calculator

the BAND class

Class for SCMSUITE BAND ASE calculator. It is imported and used the following way: eg:

```
from ase.calculators.scm import BAND

some_molecule = ase.Atoms(...)
some_molecule.set_calculator(BAND(label=some_mol))
```

Will create a directory "some_mol", in which some_mol.run will be found and executed. Note that it requires at least a label as argument.

Arguments and default values

The options passed to the constructor are the exact same ones as in the ADF calculator, wih the addition of the following one: in calculation arguments:

- `kspace = k-space` grid quality, defaults to `Normal`
  - `GammaOnly`
  - `Basic`
  - `Normal`
  - `Good`
  - `VeryGood`
  - `Excellent`

2.2.3 DFTB Calculator

the DFTB class

Class for SCMSUITE DFTB ASE calculator. It is imported and used the following way: eg:

```
from ase.calculators.scm import DFTB

some_molecule = ase.Atoms(...)
some_molecule.set_calculator(DFTB(label=some_mol))
```

Will create a directory "some_mol", in which some_mol.run will be found and executed. Note that it requires at least a label as argument.

Arguments and default values

The following options can be passed to the constructor:

template [string,] name of a .adf file containing the calculation details. Any other argument passed to the calculator will override this template.

restart: string, prefix for a restart file. May contain a directory. Default is None, so the calculations don't restart except if specified.

ignore_bad_restart_file: True or False, Ignore broken or missing restart file. By default, the scripts throw errors if the restart file is missing or broken.

label: string, base name used for all created files. May contain a directory.

atoms: ASE Atoms object, Optional Atoms object to which the calculator will be attached. When restarting, atoms will get its positions and unit-cell updated from file.

job: type of job this calculator will run, Defaults to GeometryOptimization, in which case any method like "get_potential_energy()" used on the attached atoms object will use the native ADF drivers to perform a geometry optimization. To use an ASE optimizer like BFGS or FIRE, set this option to SinglePoint.

- `SinglePoint`

- `GeometryOptimization`
- `Frequencies`
- `None, only use if template is given`

Calculation arguments: these will be used for the DFTB job construction.

- `charge = Charge` of the Atoms object, defaults to `0`
- `spin = Spin` of the Atoms object, defaults to `0` (spin unrestricted not yet implemented)
- `dftbmodel = DFTB scheme` to use in the calculation. Default is `DFTB`.
  - `DFTB`
  - `SCC-DFTB`
  - `DFTB3`
- `dftbparameters = Optional directory` for dftb parameters files. Defaults to SCM library.

2.2.4 REAXFF Calculator

the REAXFF class

Class for SCMSUITE REAXFF ASE calculator. It is imported and used the following way: eg:

```
from ase.calculators.scm import REAXFF

some_molecule = ase.Atoms(...)
some_molecule.set_calculator(REAXFF(label=some_mol, forcefield=some_forcefield.ff))
```

Will create a directory "some_mol", in which some_mol.run will be found and executed. Note that it requires at least a label as argument.

Arguments and default values

The following options can be passed to the constructor:

template [string,] name of a .adf file containing the calculation details. Any other argument passed to the calculator will override this template.

restart: string, prefix for a restart file. May contain a directory. Default is None, so the calculations don't restart except if specified.

ignore_bad_restart_file: True or False, Ignore broken or missing restart file. By default, the scripts throw errors if the restart file is missing or broken.

label: string, base name used for all created files. May contain a directory.

atoms: ASE Atoms object, Optional Atoms object to which the calculator will be attached. When restarting, atoms will get

Calculation arguments: these will be used for the ReaxFF job construction.

- `forcefield`: library to use for parameters. Note: the calculation will fail if this is not specified!

2.3 Examples

the following examples work with all calculators of the ADF modeling suite. Any python script using these libraries should be launched using the "$ADFBIN/startpython" binary.

2.3.1 Geometry Relaxation via ADF Drivers

```
# Required imports
    from ase.calculators.scm import ADF
    from ase.io              import read

    # Read molecular geometry from xyz file
    mol = read('some_mol.xyz')

    # Create calculator object with calculation details. Note the job argument
    calc = ADF(label = 'some_mol',
               job   = 'GeometryOptimization',
               basis = 'DZP',
               xc    = 'GGA:BP',
               core  = 'Small')

    # Assign calculator to system object
    mol.set_calculator(calc)

    # Due to the requested ``job`` the final energy of a geometry optimization results,
    # which is internally performed by ADF
    e = mol.get_potential_energy()
```

2.3.2 Geometry Relaxation with ASE

```
# Required imports
    from ase.calculators.scm import ADF
    from ase.optimize.bfgs   import BFGS
    from ase.io              import read

    # Read molecular geometry from xyz file
    mol = read('some_mol.xyz')

    # Create calculator object with calculation details. Note the job argument
    calc = ADF(label = 'some_mol',
               job   = 'SinglePoint',
               basis = 'DZP',
               xc    = 'GGA:BP',
               core  = 'Small')

    # Assign calculator to system object
    mol.set_calculator(calc)

    # For this ``job`` type the potential energy calculation only yields
    # a single point energy at this stage
    e_init = mol.get_potential_energy()

    # Geometry optimisation requires a driver object (here BFGS) from ``ase.optimize``
    driver = BFGS(mol, trajectory='some_mol.traj')
```

```
    # Invoce optimiser run to converge nuclear forces below 0.05 eV/Angstrom
    # within 200 steps at most
    driver.run(fmax=0.05, steps=200)
```

2.3.3 Nudged Elastic Band transition state search

For analysis of the results, see the excellent tutorials on the web : ASE examples for NEB (https://wiki.fysik.dtu.dk/ase/tutorials/neb/diffusion.html#diffusion-tutorial)

```
# Required imports
    from ase.calculators.scm import ADF
    from ase.io              import read
    from ase.neb             import NEB
    from ase.optimize        import FIRE

    # st endpoints of reaction path
    initial = read('initial.xyz')
    final = read('final.xyz')

    # create a list of Atoms objects (images) with initial and final structures at the beginning and
    # respectively, and <num_images> placeholder copies in-between
    num_images = 3
    images = [initial] + [initial.copy() for i in range(num_images)] + [final]

    # set calculators for each entry in the list. Note the SinglePoint Job type
    index=0
    for image in images:
        image.set_calculator(ADF(label      = 'neb.{0}'.format(index),
                                 job        = 'SinglePoint',
                                              basis      = 'DZ',
                                              xc         = 'LDA',
                                              core       = 'None',
                                              relativity = 'Scalar'))
        index += 1

    # form NEB object from the list.
    # The images between initial and final will then be changed during the NEB interpolation
    neb = NEB(images)

    # Interpolate linearly the positions of the middle images:
    neb.interpolate()

    # optimize the neb object with robust algorithm like FIRE:
    driver = FIRE(neb, trajectory='neb.traj')
    driver.run(fmax=0.05, steps=200)
```

A command line utility can also be called directly using the startpython binary. The following is a reproduction of the NEB run found in the ADFHOME/examples/scmlib/neb_ase_halogenexchange/ directory.

```
$ADFBIN/startpython $ADFHOME/scripting/ase/neb.py -p ADF -i initial.xyz -f final.xyz  -u "charge -1 i
```

A complete list of options can be obtained using the standard help ("-h") argument:

```
$ADFBIN/startpython $ADFHOME/scripting/ase/neb.py -h

usage: NEB [-h] -p P -i I -f F [-c] [-o] [-u U] [-m M] [--template TEMPLATE]
           [--forcefield FORCEFIELD] [--fmax FMAX] [--images IMAGES]
```

```
           [--steps STEPS] [--spring SPRING] [--logfile LOGFILE]

Nudged Elastic Band transition state search

optional arguments:
  -h, --help            show this help message and exit
  -p P                  Program that is used in the calculations. implemented:
                        BAND, ADF, DFTB, REAXFF
  -i I                  Initial geometry in ASE readable format
  -f F                  Final geometry in ASE readable format
  -c                    If used, does not implement the climbing image method
  -o                    If used, initial and final geometries are not
                        optimised
  -u U                  All options in this string will be passed litterally
                        to the ADF calculator
  -m M                  Constraint. All atoms whose symbols are passed here
                        will be fixed. ex: -mask CuAlO will fix all copper,
                        Aluminium and oxygens during optimisation
  --template TEMPLATE   Template file from adf GUI, for bells and whistles
  --forcefield FORCEFIELD
                        Force field to use in case of REAXFF
  --fmax FMAX           Maximum force limit for optimisation convergence (in
                        eV/Angstroms)
  --images IMAGES       Number of images in the interpolation
  --steps STEPS         Iteration limit for optimisation convergence
  --spring SPRING       Spring constant used between images
  --logfile LOGFILE     Logfile of the NEB calculation
```

CHAPTER

THREE

FLEXMD

FlexMD is a python library for Flexible multi-scale Molecular Dynamics simulation, developed by Rosa E. Bulo, Christoph Jacob, Stefano Borini, Tao Jiang, Jelle Boereboom, Stanislav Simko and Hans van Schoot.

3.1 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 15) [*1* (page 32)] 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 32)], opening the possibility for evaluation on time-scales beyond the reach of standard molecular dynamics simulations.

The FlexMD package is designed to make simulation options possible that are not available natively in the ADF package. Its flexible nature makes it very versatile, but comes at a cost. This cost might be completely negligible in most simulations, but it can be very high in some cases (usually when combining only cheap methods such as forcefields).

The intended users for the FlexMD package are those with some Unix/Linux experience and a basic understanding of the Python Programming Language (http://python.org/). The user is also supposed to have a basic understanding of the various methods he wishes to combine. For example, if metadynamics is supposed to be combined with ADF, FlexMD expects the user to have knowledge about DFT calculations and the usage of Collective Variables. Finally, as with every computational method, the user should monitor the FlexMD performance, both in accuracy and speed.

3.2 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

3.3 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 startpython, 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.

```
$ startpython
 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:

```
$ startpython
 from scm import flexmd
 help(flexmd)
```

Python can also give the help documentation as plain text:

```
$ startpython
 from scm import flexmd
 import pydoc
 print pydoc.render_doc(flexmd.ForceJob, "Help on %s")
```

3.4 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 32)] 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 32)], 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 32)]. 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).

3.5 Multi-scale Molecular Dynamics

The design of the **ForceJob** class allows for flexible extension of its behavior, while at the same time keeping the client code unaware of its nature: it can either act as a simple wrapper for ADF programs, or it can be a more complex orchestrating class, combining simpler **ForceJob** classes to implement multi-scale strategies. One application of this extensible design can be found in the **QMMMForceJob** object, which combines a QM and an MM method in an IMOMM-type scheme (mechanical embedding). The **QMMMForceJob** object is assigned two other **ForceJob** objects, the first representing the high-resolution calculation (QM), while the other represents the low resolution (MM). Both **ForceJob** objects contain an **MDMolecule** object for the full molecular system. The selection of the QM-region is handled by the **QMMMForceJob**, which contains the information about the part of the molecule that constitutes the QM region. When forces are requested from the **QMMMForceJob**, the following behavior is orchestrated: first, a MM force calculation is performed on the full system; then, the QM-region is selected, a QM calculation is executed solely for that region, and energy and forces are added to those from the full system MM calculation. Finally, an MM calculation is computed for the small QM-region, and the energy and forces are subtracted, yielding the final result, returned to the invoker. In symbols:

EQM/MM(Full) = EMM(Full) + EQM(QMRegion) – EMM(QMRegion)

The **QMMMForceJob** handles periodic boundary conditions if the low-level (MM) method supports this feature (i.e. NAMD). Whether the periodic interaction of the QM region with itself is handled at high or low resolution depends on the method used for the QM calculation. An example of QM/MM MD calculations can be found in the examples directory examples/scmlib/qmmm_dftbUFF_2h2o. The **QMMMForceJob** allows the use of link atoms when the QM boundary cuts through covalent bonds. However, this feature comes at the price of an increased script complexity. An example of a QMMM link-atom MD simulation is provided in the examples directory, under examples/scmlib/qmmm_linkatom_dftbNAMD_glutamate.

For more complex multi-scale calculations the **HybridForceJob** class can be used. This class allows the combination of a large set of different **ForceJob**s, each of them describing either the same, or different molecular systems. Each **ForceJob** can either involve a calculation on the full **MDMolecule** object it contains, or restricted to a specified region of the corresponding molecule. The forces from each contributing **ForceJob** can either be added or subtracted from the total force according to user preference, as specified at construction of the **HybridForceJob** object.

In order to perform QM/MM simulations on chemical reactivity in solution, it is important that the description of the solvent molecules can change on the fly, as the molecules move towards or away from the reactive region. To facilitate this, an **AdaptiveQMMMForceJob** class is available to provide adaptive QM/MM simulations using several available schemes, as described by Bulo et al.[4 (page 32)] 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 32)] method, is provided in the examples directory, examples/scmlib/adqmmm_mopacscmUFF_h2o.

3.6 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 32)] and umbrella sampling [6 (page 32)]. 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 32)].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.

3.7 Working with FlexMD

It is recomended to read the sections *Introduction* (page 27) and *Molecular Dynamics* (page 27) before working with FlexMD. Basic understanding of the Python Programming Language (http://python.org/) is also required. The Python website hosts documentation and a tutorial (http://docs.python.org/2/tutorial/) that can be used to learn Python.

The performance of the FlexMD package is difficult to predict because it depends on system size, the type of ForceJobs used and how these ForceJobs are combined. It is advised to first test the overhead of the FlexMD package for your system before running large simulations. When ab initio forces are involved, the overhead should not give a significant performance penalty. However, it may become a bottleneck when your system only uses cheap forcefields.

3.7.1 Creating a molecule object

FlexMD can be run trough the interactive python interpreter in the ADF package. To start it, run: $ADFBIN/startpython 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 $ADFBIN/startpython 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 $ADFBIN/startpython, 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 automaticly 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)
```

3.7.2 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 $ADFHOME/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,'-
```

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 $ADFHOME/examples/scmlib.

3.7.3 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 32)] for this. The MDPropagatorJob object is created just like the other objects in FlexMD:

```
   # do this after importing flexmd and creating a ForceJob.
   # it creates the MDPropagator job, with some settings
myMDJob = flexmd.ASEMDPropagatorJob( ForceJob=myForceJob )
myMDJob.settings.set_tempcontrol( True, nhfreq=2, maxdef=50.0 )
myMDJob.settings.set_temperature(300.0)
myMDJob.settings.set_timestep( 0.02 )
```

For more details on the ASEMDPropagatorJob, view it's help page: help(flexmd.ASEMDPropagatorJob), or take a look at the MDSettings object: help(flexmd.MDSettings).

The propagatorJob can be used to create an MDManager object:

```
   # create an MD manager
myManager = flexmd.MDManager( mdjob=myMDJob)
```

The manager object is now in control of the MD simulation, and we can use it to run the simulation for a number of steps:

```
   # tell the MD manager to run the simulation
myManager.run( ncycles = 2500 )
```

Note that the number of steps here should be increased a lot if metadynamics effects are to be observed, but it is always wise to first run a small number of steps to check if everything works. Some information will be printed during the simulation, depending on the settings of the components used. The manager will also create some folders in the working directory, and store the data produced by the simulation in there.

The full flexmd jobfile should now look something like this:

```
from scm import flexmd
myMol = flexmd.MDMolecule('myGeometryFile.xyz')
myMol = flexmd.pdb.set_box([50.0,25.0,100.0])
   # setup our reaxff ForceJob and attach the forcefield file
   # (place the ff file in the same dir as the script and the xyz!)
myReaxffForceJob = flexmd.ReaxffForceJob(molecule=myMol)
myReaxffForceJob.settings.set_ff_filename('reax_forcefield_file.ff')

   # next we define the collective variable: the distance between atom 1 and 2
myCvs = [flexmd.DistCV([1,2])]
   # create a set of metadynamics properties, using the CV
mtdSettings = flexmd.MetadynamicsSettings(cvs=myCvs, widths=[0.30], height=0.25 )
   # create the metadynamics job by combining the molecule, settings (with CV)
   # and the number of md steps between depositing metadynamics hills.
myMetadynamicsjob = flexmd.MetadynamicsJob( myMol, settings=mtdSettings, nstep=150 )

   # add a wall to prevent the two atoms from drifting more than 10 Angstrom away.
myWalljob = flexmd.WallJob(molecule=myMol, cvs=myCvs, cntrs=[10.0], widths=[1.0], heights=[500.0])

   # combine the forces into a hybrid job that will be used for the MD
myForceJob = flexmd.HybridForceJob( [[myReaxffForceJob,'+'], [theMetadynamicsjob,'+'], [theWalljob,'-

   # 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 )

   # create an MD manager
myManager = flexmd.MDManager( mdjob=myMDJob)
   # tell the MD manager to run the simulation
myManager.run( ncycles = 2500 )
```

3.8 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 2014, SCM, R. E. Bulo, C. R. Jacob, S. Borini, A python library for flexible multi-scale molecular dynamics simulations. http://www.scm.com

3.8.1 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.

3.9 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) (http://dx.doi.org/10.1109/5992.998641). The Atomistic Simulation Enviroment website and documentation (https://wiki.fysik.dtu.dk/ase/)
2. C.R. Jacob, S.M. Beyhan, R.E. Bulo, A. Severo Pereira Gomes, A.W. Gotz, K. Kiewisch, J. Sikkema, L. Visscher, *PyADF - A scripting framework for multiscale quantum chemistry.* J. Comput. Chem. 32, 2328-2338 (2011) (http://dx.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) (http://dx.doi.org/10.1007/s00894-004-0183-z)
4. R.E. Bulo, B. Ensing, J. Sikkema, L. Visscher, *Toward a Practical Method for Adaptive QM/MM Simulations.* J. Chem. Theory Comput. 5, 2212-2221 (2009) (http://dx.doi.org/10.1021/ct900148e)
5. A. Laio, M. Parrinello, *Escaping free-energy minima.* Proc. Natl. Acad. Sci. USA., 99, 12562-12566 (2002) (http://dx.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) (http://dx.doi.org/10.1016/0010-4655(95)00053-I)
7. M. Bonomi, D. Branduardi, G. Bussi, C. Camilloni, D. Provasi, P. Raiteri, D. Donadio, F. Marinelli, F. Pietrucci, R.A. Broglia, M. Parrinello, *PLUMED: A portable plugin for free-energy calculations with molecular dynamics.* Comp. Phys. Comm. 180, 1961-1972 (2009) (http://dx.doi.org/10.1016/j.cpc.2009.05.011). PLUMED website and documentation (http://www.plumed-code.org/)

CHAPTER

FOUR

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

4.1 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

4.1.1 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.