pydlpoly Documentation

Release 1.0

Rochus Schmid

February 02, 2016

Contents:

CHAPTER

ONE

INTRODUCTION OF PYDLPOLY'S GENERAL CONCEPTS

1.1 Motivation

The `pydlpoly` project was initiated out of a need. Before we had used the `Tinker` code (V 4.2) which was not parallel and limited to about 3000-5000 atoms. We looked at `lammps` and `DL_Poly 2`. Because of the simpler structure and the fact that `DL_Poly 2` was not using a domain decompostion to be data-parallel we chose it as a basis for wrapping the code with `f2py`. Since all the data like atom positions, forces, velocities etc. are available on all the nodes the code is limited to about 30,000 atoms, which is still an order of magnitude more than what was possible for us with `Tinker`. Our target is not so much to investigate larger systems but to improve the potential energy functions for the hybrid systems we are interested in (like MOFs). We are working on polarizable and "semi-reacive" force fields, which means that the computational cost per atom will get larger in the future. We initalliy started with V2.20 of `DL_Poly`, which was later converted and renamed to `DL_Poly Classsic`, distributed under BSD-license. `pydlpoly` is a fork of `DL_Poly CLassic` and is currently available to academic groups on the basis of a "collaborational license".

1.2 Features and Missing Features

The idea behind `pydlpoly` is to use the F90 (with a lot of F77 flair :-) MM energy engine as a backend and to do all the operations like geometry optimization, MD startup, analysis on Python level. Few things have been added into the F90 code, whereas most extensions and features are implemented on the scripting level. Some things suffer from efficiency issues on Python level and will have to be converted to F90 at some point. Please note that `pydlpoly` is a growing and evolving project (as is this documentation ... it will always be outdated). A very important point is that `pydlpoly` has been developed with our projects in mind. This means it supports all we needed to do for our work and probably a number of things are missing. Most importantly `pydlpoly` – even though it is extended in severel ways by the Python frontend – must be seen as a "cripled" `DL_Poly Classic` since all the features implemented in `Dl_Poly Classic` which were not needed might be broken. This means some of these features (like hyperdynamics, solvation, metal potentials etc.) might still work in the remaining F90 routines (some might not) but none of these things are supported by the Python frontend. If you want to use these features you are probably better off if you use the `DL_Poyl Classic` code itself.

1.2.1 Changes/Additions to `DL_Poly` (F90-level)

- vdW lookup tables use cubic spline interpolation and forces are calculated from analytic derivative of cubic spline to be consistent with energy (original `DL_Poly 2` uses interpolation for energy and force giving inconsistent forces leading to non energy conserving microcanonic MD).
- vdW and short ranged Coulomb (either truncated with shift damping or the short range part of Ewald/SPME) are switched to give continuous forces at the cutoff.

- Gaussian charges have been added for all types of Coulomb interactions (truncated, Ewald and SPME) with a different width for each atom.
- A number of potentials have been added like MM3 type bond and stretch terms, four fold torsion or Feynman-Hibbs potentials force vdW interactions.
- Switching of non-bonded interactions (vdW and Coulomb) on a per molecule basis is added (also kinetic energy can be switched off on a per molecule basis).

1.2.2 Features

- Access to all internal data structures like positions, forces, velocities.
- Efficient L-BFGS optimizer for molecular structures and a robust minimzer for periodic systems was added.
- Energy calculations are performed from a central Python level function, which allows to add arbitrary energy contributions from other Python modules (like QMMM etc)
- The central MD loop is controlled on Python level, which allows complex scripting.
- The input for the underlying `DL_Poly` engine (FIELD, CONFIG, CONTROL) are generated completely automatic from input files inspired by the `Tinker` format. In particular, the force field is defined using atom types in a general force filed file definition (key-file) which is parsed to produce the FIELD input
- Setup allows to automatically add e.g. guest molecules to a periodic system or to fill a box with solvent molecules.
- Simple mechanic embedding QMMM (currently with `TURBOMOLE` as QM engine).
- Restart and trajectory information is written to a hdf5 restart file allowing complex scripting by grouping the data in "stages"

1.2.3 Missing Features (`DL_Poly` features not supported any more)

- Only Velocity Verlet but no Leap frog propagator from originial code
- Many special potentials (metal potentials etc.) not supported by frontend
- Hpyerdynamics and solvation features not supported
- Metadynamics implemented in `DL_Poly Classic` is not available. This can be replaced by an experimental coupling to the `plumed` metadynamics library (we have written a `f2py` wrapped `pyplumed` for this)
- probably more I am not even aware of ...

1.3 Dependencies

For details see the Installation section. This is just an overview over the major libraries one needs for `pydlpoly`

- Python 2 (not 3!!) .. I guess you need at least 2.5
- Numpy (which includes `f2py`)
- mpi4py (Cython based MPI wrapper ... we use OpenMPI)
- h5py (wrapper for hdf5 library)
- fftw2 (`DL_Poly` is interfaced to fftw2 only and not to fftw3)

1.4 Status and Outlook

This project is constantly developing. Currently we are working on QEq based polarization models. Most importanly the force field paramter fitting, which was built on the `Tinker` engine is currently replaced by completely new pythonic ff_generator using `pydlpoly` as MM engine.

1.5 Contributors

Concept, Idea and most of the work by R. Schmid Further contributions by

- Chris Spickermann (Gaussian charges, inital QEq)
- Mohammad Alagehmandi (initial versions of GCMD)
- Johannes P. Dürholt (ff_gen infrastructure, addtions to QM/MM)
- Niklas Siemer (FH-potentials)

CHAPTER

TWO

COMPILING AND INSTALLING PYDLPOLY

2.1 Getting the code

This assumes you are a member of the CMC-group or have at least read access to our main repository, which lives on Rochus's workstation legolas.aci.rub.de. Get yourself `mercurial` installed (or better use `TortoiseHg`, which is a very nice GUI to mercurial) and clone the repo like this:

```
cd
mkdir sandbox
cd sandbox
hg clone ssh://legolas.aci.rub.de//home/repo/repos/pydlpoly
```

This should produce the pydlpoly source files into the `~/sandbox/pydlpoly` directory.

If you are not a CMC-group member, you probably have tgz file to unpack. For the further discussion we assume you do this in the same directory.

2.2 Generating the Docu

For building the docu (these files) you need to have the `sphinx` documentation system installed. On Ubuntu variants it is usally the package `python-sphinx` which needs to be installed. Just go into `doc/` and `make html`. To open open `doc/_build/html/index.html` in your favorite browser.

2.3 Compiling the code

2.3.1 Dependencies

As overviewed in the Introduction (*Dependencies*) the compilation of `pydlpoly` depends on a number of Python packages. First you need a F90 compiler. We have used both `gfortran` and `ifort` on our machines, but others will work as well. Then you need the development packages (libs&includes) of Python and Numpy. Numpy contains `f2py`, which is needed to generate the F90/Python bindings. On Ubuntu and derived distributions it is often sufficient to use the package manager to install the Python wrapper packages in order to install also the underlying libraries itself.

- `mpi4py` : The MPI bindings for Python - in addition the MPI library incuding the development options will be needed. We prefer to use OpenMPI.
- `h5py` : The hdf5 library bindings for Python - in addition the hdf5 lib incuding development options will be installed

- `fftw2` : You need the Fastest Fourier Transform in the West Version 2 (not 3!). On some distributions only fft3 is available as a package. In this case you need to compile it manually. You have at least to add `-fPIC` because *f2py'* will generate a shared object loaded at runtime.
- `blas` : You need a `libblas` to get linked in. This is not an extremly critical lib in terms of performance so any "vanilla" BLAS-library can be used, as long as it is compiled `-fPIC`.

2.3.2 Makefile

The F90 sources, the `f2py` interface definition (_pydlpoly.pyf) and the Makefile live in the `source/` directory. Depending on the position of the libraries you might have to adapt the Makefile. The following is our standard Makefile with two sections, one for the typical LINUX worstation (using gfortran) and a special variant on our cluster (hostname cmcc), where all packages (including Python/Numpy, OpenMPI, hf5 etc.) are manually compiled with `ifort` and using MKL.

```
FC=mpif90
LD=mpif90 -o

ifneq ($(HOSTNAME),cmcc.cmcc.cluster)
#================== GNU Fortran, MPI version ============================
# gfortran (SUSE version , fftw2 is called libdfftw
LDFLAGS=-O2 -ffast-math
FFLAGS=-c -O2 -ffast-math -fPIC
LDLIBS=-L/usr/local/lib -lfftw -lblas
F2PY_COMP=gnu95
else
## cmcc version using ifort and fftw and MKL Blas
LDFLAGS=-O3 $(LDLIBS)
LDLIBS=-L/usr/local/lib -lfftw
FFLAGS= -FI -c -O3 -xhost -fPIC
F2PY_COMP=intelem
endif

#=======================================================================
```

It is important to set the `F2PY_COMP` variable to the proper value for your F90 compiler. Check the manual and help of `f2py` to get this right.

2.3.3 Compiling and Installing

If your Makefile is properly configured a simple `make` in the `source/` directory will compile all the fortran code, generate the binary _pydlpoly.so, which will be copied to the `so/` directory. If this fails you have most probably a dependency problem. In this case please check the error message to fix it. In order to run the code you need to add the two directories `py/` and `so/` to your `PYTHONPATH` environment variable. Usually this is doen in .bashrc (provided you use `bash`) as your shell.

```
export PDLPDIR=/home/rochus/sandbox/pydlpoly
export PYTHONPATH=$PDLPDIR/so:$PDLPDIR/py:$PYTHONPATH
```

To test if everything works allright just import the `pydlpoly` module in your interactive Python session like this:

```
Python 2.7.5+ (default, Feb 27 2014, 19:37:08)
[GCC 4.8.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import pydlpoly
```

```
Intialized Random with System Time
>>>
```

If there are no error messages you are ready to go!

CHAPTER

THREE

FORMAT OF PYDLPOLY'S INPUTFILES

The native input files `CONFIG`, `FIELD` and `CONTROL` read by the core `DL_Poly` engine are still produced on the fly by the Python frontend and can be checked in the temporary rundirectory. However, the main input files are the <key> and <xyz> file, which contain the force field definition and the geometry/connectivity. The are inspired by the Tinker format, but especially the <key> file format is not exactly equal to Tinker and not compatible to it.

3.1 The Geometry file (extension .xyz)

Here are the first lines of the file mof5.xyz describing the periodic MOF called MOF-5:

```
 424    26.0699    26.0699    26.0699    90.0000    90.0000    90.0000
 1 o      0.003061   -0.000459   -0.000450   165      2      3      4      5
 2 zn     1.139801   -1.137243   -1.137311   166      1     14     18     21
 3 zn    -1.133735   -1.137131    1.136441   166      1      6     15     23
 4 zn    -1.134000    1.136288   -1.136999   166      1     11     12     17
 5 zn     1.140090    1.136351    1.136079   166      1      8      9     20
 6 o     -0.796133   -0.799964    3.041162   167      3      7
 7 c      0.003076    0.000224    3.621042   168      6      8    230
 8 o      0.802262    0.800295    3.040974   167      5      7
 9 o      0.804535    3.041271    0.797591   167      5     10
10 c      0.003073    3.621184   -0.000322   168      9     11    210
11 o     -0.798196    3.041106   -0.798310   167      4     10
.
.
```

Appart from the unit cell parameter information in the first line, this file is exactly identical to the regular Tinker xyz format and can be opened by `molden` or `vmd`.

3.1.1 First line

- The first integer defines the number of atoms (and thus following lines in the file)
- The next three floats define the a, b and c axis length of the unit cell in Angstrom.
- The last three floats define the cell angles alpha, beta and gamma in degree.

If the system is non-periodic the cell parmaters must be omitted (Warning: `molden` writes Tinker xyz files with some text after the number of atoms. You need to remove this before reading with `pydlpoly`).

3.1.2 Further lines

- The first integer number is just a running index starting from 1! (not really clear why but we kept it to keep the format readable by `molden`)
- The second entry is the element
- The next three floats (col 3-5) are the xyz cartesian coordinates in Angstrom
- The next entry (col 6) is the atom type. `pydlpoly` handles these as strings and you can have any name (like "Zn_MOF5" or so). But for compatibility with `molden` or `vmd` the atom types must be numbers as required in Tinker. Alternatively, an `#atomtypes` flag can be appended at the end of the xyz file to provide a list of strings, numbers and elements and allow `pydlpoly` to work with a mixed input from xyz- and *The Force Field file (extension .key)*. Example:

```
#atomtypes
165 O_Zn_MOF5 o
166 Zn_MOF5 zn
167 O_C_Zn_MOF5 o
168 C_MOF5 c
```

- All further integers are the indices of the connected atoms (defined back and forth .. so 1 binds to 2 and 2 to 1)

3.2 The Force Field file (extension .key)

The force field parameters are defined in the key file, which is derived from the original Tinker format. More recently we have abandonned the backwards compatibility to Tinker and thus some obsolete information was discarded from the key files. We refer to these files as "version 2.0" files. The script `convert_key` can be used to convert the old file format into the new format. Note that the current `pydlpoly` can **only** read the new format.

The original Tinker key-file format is described in detail on the Tinker website.

The pydlpoly specific key file format is described below. The file is organized in lines starting with a keyword. Comment lines start with a hash tag "#". There are settings keywords which should appear only once (last occurance will overwrite previous settings) and potential term keys. The latter define a specific potential type. Potential type keywords are always followed by one or more potential type specifiers. Originally these potential types were integer numbers (derived from the "ancient" MM3 atom types). However, the potential types do not need to be integer numbers but can also be strings (see also *The Geometry file (extension .xyz)*) for better readability. The drawback is that in this case the written xyz files are not readable by `molden`. This problem can be circumvented by setting the `moldenr` flag of the `pydlpoly.pydlpoly.write_tinker_xyz` method to `True`, which automatically replaces the strings with numbers. For historic reasons, the examples shown below still use integer atom types.

Here is an example for the `MOF-FF` (see this paper) force field for MOF-5:

```
version         2.0
parameters      none

bondunit        71.94
angleunit       0.02191418
strbndunit      2.51118
opbendunit      0.02191418
torsionunit     0.5
vdwtype         exp6_damped
vdwdampfact     0.25
radiusrule      arithmetic
radiustype      r-min
radiussize      radius
```

```
epsilonrule     geometric
a-expterm       184000.0
b-expterm       12.0
c-expterm       2.25
bondtype        mixmorse_bde
strbndtype      mmff
opbendtype      mmff
chargetype      gaussian

# params for phenyl group
atom            5         h
atom            2         c

vdw             5              1.5000      0.0200
vdw             2              1.9600      0.0560

bond            2         2             7.0800      1.3940
bond            2         5             5.4300      1.0940

angle           2         2         2             0.7410    127.0500
angle           2         2         5             0.5030    120.3500

torsion         2         2         2         2             0.0000      4.3790      0.0000
torsion         5         2         2         5             0.0000      5.9720      0.0000
torsion         2         2         2         5             0.0000      6.3160      0.0000

opbend          2         2         5         2             0.0190

strbnd          2         2         2             0.0470      0.0470      0.4990
strbnd          2         2         5            -0.1750      0.3720      0.6490

charge          5              0.1200      0.7236
charge          2             -0.1200      1.1630

# params for Zn4O unit
atom            165       o
atom            166       zn
atom            167       o
atom            168       c

vdw             165            1.8200      0.0590
vdw             166            2.2900      0.2760
vdw             167            1.8200      0.0590
vdw             168            1.9400      0.0560

bond            165       166           1.4890      1.9870     50.0000
bond            166       167           1.6650      1.9170     50.0000
bond            168       167           8.6320      1.2750
bond            2         168           4.9370      1.4880

angle           166       165       166             0.6980    103.9920
angle           165       166       167             0.0000    113.5840
angle           167       166       167             0.0800    123.1030
angle           166       167       168             0.0940    135.6060
angle           167       168       167             1.5550    123.0060
angle           2         168       167             1.0730    116.3680
angle           2         2         168             0.8060    117.2960
```

```
torsion        166       165       166       167             0.0000      0.0000      0.0000
torsion        165       166       167       168             0.0000      0.0000      0.0000
torsion        166       167       168       167             0.0000      0.0760      0.0000
torsion        166       167       168       2               0.0000      3.0120      0.0000
torsion        167       166       167       168             0.0000      0.0000      0.0000
torsion        2         2         168       167             0.0000      1.9020      0.0000
torsion        2         2         2         168             0.0000      0.0000      0.0000
torsion        5         2         2         168             0.0000      0.0000      0.0000

opbend         168       167       2         167             0.1880
opbend         2         168       2         2               0.0880

charge         165            -1.7800      1.1176
charge         166             1.4200      2.0733
charge         167            -0.7200      1.1176
charge         168             0.6100      1.1630
chargemod      2         168             0.1800      1.1630

strbnd         166       165       166            0.1280      0.1280      0.0530
strbnd         165       166       167            0.1960      0.0400     -0.1610
```

3.2.1 Settings Keywords

Note: A remark on the units used in pydlpoly: all energies are given in **kcal/mol**. `Dl_Poly` allows differnt energies to be used but by default a `kcal` is written to the `FIELD` file. Note that `DL_Poly` internally works with a unit of 10 J/mol and internal energies are converted to kcal/mol in `pydlpoly` using `dlp.engunit`. However, for historic reasons our key files use the MM3 convention for force constants (as Tinker usually does) using mdyne/A.

- bondunit: Factor to convert force constants in bond keyword lines to kcal/molA^2 (also contains 0.5!!)
- angleunit: dito for angles
- strbndunit: dito for strbnd terms
- opbendunit: dito for out -of-plane bend terms
- torsionunit: dito for torsions (we use barriers in kcal/mol as in Tinker so factor is 0.5)

Note: These factors also depend on the (partly strange) definition of potential terms in Tinker (we have adopted these historically for compatibility). Please check the numbers in the `FIELD` file in case you are not sure what is really used in the calcualtions.

- vdwtype: defines the type of vdw term used (`exp6_damped`: MOF-FF specific modified Buckingham, `buckingham` or `lennard-jones`)
- vdwdampfact: only needed in case of `exp6_damped`
- radiusrule: combination rule (either `arithmetic` or `geometric`)
- radiustype: either `r-min` or `sigma`
- radiussize: either `radius` or `diameter`
- epsilonrule: combination rule for epsilon (either `arithmetic` or `geometric`)
- a-term/b-term/c-term: specific params for the MM3 type `buckingham` potential (also valid for `exp6_damped`)

- bondtype: can be omitted, if `mixmorse` or `mixmorse_bde` is used then the morse potential is employed for any bond term with three instead of two paramters (see below)
- strbndtype: for MOF-FF request `mmff` type
- opdendtype: for MOF-FF request `mmff` type
- chargetype: using `gaussian` here switches to Gaussian type charge distributions (changes format of `charge` and `chargemod` potential keyword format)
- rigid: list of molecules (defined e.g. via molname) to be kept as rigid units (works only for MD!)
- freeze: list of molecules to be kept frozen

3.2.2 Potential Keywords

This table is giving an overview about potential keywords (they can show up multiple times). please see the comments at the end of the table and in particular the examples in the next subsection.

Keyword	# atoms	Parameter 1	Parameter 2	Parameter 3	Parameter 4	Comments
atom	1	element				defines a type
vdw	1	radius	epsilon			use combination rules
vdwpr	2	radius	epsilon			explicit params for a pair
bond	2	force-const	refdist	[alph/bde]		mixmorse: use morse pot if third paramter is given.
bond5	2					
angle	3	force-const	refangle			
angle5	3					
anglef	3					
anglef-2	3					
torsion	4	V1	V2	V3	[V4]	fourth is optional
torsion5	4					
opbend	4	force-const				first atom is central
strbnd	3	strbnd1	strbnd2	strstr		refval from bond and angle
charge	1	q	[sigma]			sigma for gaussians
chargemod	2	q	[sigma]			modifies charge
virtbond	2					no params
molname	n	name	atom-types			name is a string!
restrain-distance	2					
restrain-angle	3					
restrain-torsion	3					
equivalent	2					atom types are equal

- aph/bde : for `mixmorse` the third param triggers the use of a morse potential with the given alpha. for `mixmorse_bde` the third param is the bond dissociation energy in kcal/mol
- torsion : barriers of torsions with n= 1, 2, 3 and optional 4 in kcal/mol
- charge : if atomtype is a **negative** integer it refers to the atom index (as in Tinker)

- chargemod : modifies charge if atom is bonded to a specfic other atom type
- virtbond: one of the two atoms must be a virtual atom (element "Xx"), no params needed
- molname: can have an arbitrary number of atomtypes and defines the name for such a system

3.2.3 Examples

- **chargemod**: Assume "Car" is the atomtype of an aromatic system which is usually -0.12. However, if "Car" is the alpha-carbon of a carboxylate (bonded to a "Ccarbox") you want it to have the same params as all other "Car" but its charge should be 0.0. For pointn charges (no sigma you can define this accordingly:

```
charge     Har             0.12
charge     Car            -0.12
chargemod  Car   Ccarbox   0.0
```

- **molname**: You want to have all benzene molecules to be named "bz". The are built from atomtypes "Cbz" and "Hbz":

```
molename bz  Hbz  Cbz
```

- **equivalent**: In your keyfile you have params for Car and Har and you want to "reuse" them with another name, for example to define a molecule name with it or to keep all params and just change a specific thing (the charge or a bond term etc.) you can make two types equivalent. The first atomtype is the new "alias" and the second mus be defined already. As long as no specific potential type for the new "alias" is found it will be replaced by the already existing params. This means an alternative way of defining the charges not using chargemod whould be this:

```
charge     Har             0.12
charge     Car            -0.12
equivalent Caralp   Car
charge     Caralp          0.0
```

  For the above definition of the benzen molname one could use the following equivalence:

```
equivalent   Hbz   Har
equivalent   Cbz   Har
```

3.3 The obsolete CONTROL File

A number of paramters in the original `DL_Poly` are set in the `CONTROL` file. This is not needed, but can be provided. If `control=<Filename>` is used as a named paramter in `pydlpoly.setup` then instead of the default `CONTROL` file the provide file named <Filename> is used. However, a large number of settings in the `CONTROL` file are no longer read and used. For example the number of MD steps or the temperature or pressure, which is directly set in `pydlpoly.MD_init`. On the other hand a number of settings like the cutoff radius or the specific way to compute the electrostatics can still be changed via the `CONTROL` files. The most direct way to do this is via the `control` directory structure of the class `pydlpoly.pydlpoly`. Before calling setup (this is important, since during setup the `CONTROL` file is generated) just change or add settings to this directory. Here is an example how to reduce the default cutoff from 12.0 to 10.0 A.:

```
import pydlpoly

pd = pydlpoly.pydlpoly("test")
pd.control["cut"] = 10.0
pd.setup()
```

It is recommended to check the final `CONTROL` file. For details about the settings see the `DL_Poly` documentation.

Todo

We need a list of CONTROL keywords which can be changed.

3.4 Naming of Molecules

3.5 Restarting

CHAPTER

FOUR

SIMPLE USAGE EXAMPLES FOR PYDLPOLY

4.1 Optimization

4.1.1 Load MOF-5 and optimize it

4.2 MD Examples

4.2.1 Load MOF-5 with benzene

4.2.2 Perform a NsT simulation with benzene@MOF-5

CHAPTER

FIVE

DOCUMENTATION OF PYDLPOLY'S API

5.1 Starting up pydlpoly

In order to start up pydlpoly you have to import the module *pydlpoly* and instantiate an object ot the pyldpoly class like this:

```
import pydlpoly
pd = pydlpoly.pydlpoly("test")
```

This generates a still "empty" pydlpoly object *pd* with the default runname "test". The default runname is used as a default for input file prefixes etc..

class pydlpoly.**pydlpoly**(*name*, *instances=None*)

__init__(*name*, *instances=None*)

startup dlpoly using name as a general runname just starting up ... no system loaded, yet.

REMARK: if instances is not None, but an integer value, we will try to generate as many independent instances of pydlpoly. We assume that the number of available nodes is an integer multiple of the number of instances, because otherwise no proper loadbalancing is possible. The code exits with an error if this is not the case

setup(*key=None*, *control=None*, *xyz=None*, *path=None*, *local=False*, *molecules=None*, *bcond=None*, *QMMM=None*, *empty_box=None*, *restart=False*, *pdlp=None*, *read_stage='default'*, *start_stage='default'*, *velocities=False*, *pdlp_restart=None*, *do_first_energy=True*, *smallrings=False*, *keep=False*, *vdw_srcut=None*, *split_vdw=False*)

Setup method to read input files of the system (or restart) and to initalize pydlpoly

Parameters

- key (str) : filename of key file (if *None* <runname>.key is used)
- control (str) : filename of control file (if *None* default is used)
- xyz (str) : filename of xyz file (if *None* <runname>.xyz is used)
- path (str) :
- local (bool): default: False - generate a subdir from runname and run job there (True - run local)
- molecules (list): add guest molecules
- bcond (int) : enforce boundary conditions (1: cubic, 2: orthorombic, 3: triclinic, 6: xy-periodic)
- QMMM

- empty_box
- restart
- pdlp
- read_stage
- start_stage
- velocities
- pdlp_restart
- do_first_energy (bool): default True - compute a first energy after reading the system in
- smallrings (bool): default False - skip detecting small rings (4, 5 rings) for e.g. angle5, bond5 types
- keep (bool): Keep raw DL_Poly files (CONFIG, CONTROL, FIELD, OUTPUT), default=False
- vdw_srcut (float): default is None - if a float is specified this is the cutoff energy in kcal/mol above which vdw rep is replaced by an inverted parabola
- split_vdw (bool): splitting vdw into dispersive and repulsive part (for seperate scling in GCMD) default: False

---

Note: By default a standard CONTROL file is generated when calling `pydlpoly.setup` which is generated using a dictionary of the *pydlpoly* class. The default entries are the following:

```
"timestep" : 0.001
"cut"      : 12.0
"delr"     : 1.0
"shift"    : "damping"
```

In order to change these settings you can directly change these values or add other options. Not all keywords from DL_Poly classic are still used in pydlpoly (like for example temp or pressure are set in *MD_init*). In order to use a 2 fs timestep and a 10.0 A cutoff you have to change the values *before* calling *setup* like this:

```
pd.control["cut"] = 10.0
pd.control["timestep] = 0.002
```

A second way is to specifiy an explcit control file in setup using the *control* parameter.

Warning: Internally the *pydlpoly* class imports the *_pydlpoly.so* library, which contains the python wrapped F90 code. All the variables like xyz coordinates etc. are held in allocatable F90 module level arrays, which exist only once (like common blocks). Therefore it is not possible to generate two instances of the `pydlpoly.pydlpoly` class, since they would share the same arrays. To prevent this an error is raised if you try to generate a second *pydlpoly* incarnation.

5.2 Optimization Methods

The following methods should be used for the structure optimization (periodic or nonperiodic) or the optimization of lattice and structure.

class `pydlpoly.`**`pydlpoly`**(*name*, *instances=None*)

LATMIN_sd (*threshlat*, *thresh*, *lat_maxiter=100*, *maxiter=1000*, *fact=0.002*, *maxstep=3.0*)
Lattice and Geometry optimization (uses MIN_lbfgs for geom opt and steepest descent in lattice parameters)

Parameters

- threshlat (float) : Threshold in RMS force on the lattice parameters
- thresh (float) : Threshold in RMS force on geom opt (passed on to `pydlpoly.MIN_lbfgs`)
- lat_maxiter (int) : Number of Lattice optimization steepest descent steps
- fact (float) : Steepest descent prefactor (fact x gradient = stepsize)
- maxstep (float) : Maximum stepsize (step is reduced if larger then this value)

MIN_lbfgs (*thresh*, *maxiter=None*, *m=10*, *nlst_rebuild=10*)
Minimize the systems energy by a L-BFGS optimizer (no lattice optimization)

Parameters

- thresh: (float) Threshold of RMS Gradient in kcal/mol/A
- maxiter: (int) Maximum Number of iterations
- m: (int) Number of previous gradients used in L-BFGS
- nlst_rebuild: (int) Number of steps until a rebuild of neigbor list is enforced

---

Note: The force used in the LATMIN optimization is symmetrized according to the boundary conditions. Thus, if for example *bcond = 1*, which means cubic boundary conditions the stress tensor is symmetrized to be diagonal with averaged values on the diagonal. Make sure to use proper boundary conditions. In order to allow a cubic system to become tricilinc you need to explcitly set *bcond = 3* in `pydlpoly.setup`.

5.3 Molecular Dynamics Methods

The MD simulations (starting form the current coordinates) are setup by two methods. The MD parameters are setup up by *MD_init* and the simulation itself is run with *MD_run*. You can use mutliple *MD_run* calls to just continue after doing something like storing the current coordinates or whatever you like. Note that the same can be accomplished by passing a function *do_every_step* to *MD_run*.

The following ensemble types are available:

- *nve* : Microcanonic Ensemble (no Thermostat/Barostat)
- *nvt* : Canonic ensemble with Thermostat, either *hoover* (Noose-Hoover) or *ber* (Berendsen)
- *npt* : NPT with isotropic pressure (for liquids with cubic bcond)
- *nst* : NsigmaT with non-isotropic pressure (make sure to have bcond = 3)

class `pydlpoly.`**pydlpoly** (*name*, *instances=None*)

MD_init (*stage*, *T=None*, *p=None*, *startup=False*, *ensemble='nve'*, *thermo=None*, *relax=None*, *traj=None*, *rnstep=100*, *tnstep=100*, *grlmb=False*, *grlmb_N=0*)
Sets up everything for a new MD stage (you MUST provide a new stage name here)

Parameters

- stage (str) : name of the new stage to be used in restart

- T (float) : temperature to be used for the thermostat (and for the intial veldist) [K]
- p (float) : pressure (in dlpoly units [kAtm]??check this??)
- startup (bool) : if True use a Maxwell-Boltzmann distribution for the velocities given by T
- ensemble (str) : type of ensemble (nve, nvt, npt, nst)
- thermo (str) : "ber" or "hoover" (includes barostat .. dlpoly manual)
- relax (tuple or list of floats) : relaxation times (taut, taup) in [ps] , taup only for npt, nst
- traj (list of str) : list of keys to keep in the trajectory file
- rnstep (int) : number of steps after which restart info is written
- tnstep (int or list of int) : number of steps after which traj info is written (if list it needs to be of the same length as traj)
- **grlmb (*False* or "added" or molecule name or list of `pdlpmol.pdlpmol` objects)** [if this is not *False* the] self.added_mols will be switched 1 during the MD
- **grlmb_N (int)** [if grlmb is "added" or a molecule_name etc. and grlmb_N > 0 (not the default) then only the first] grlmb_N molecules are grown in and the others are kept at lambda=0

Note: Also during MD in the NPT or NST ensemble one has to make sure to use the proper boundary condition with *bcond*. See the corresponding note for the Optimization methods.

5.4 Advanced Methods

the following methods can be used in various scripts to do more advanced things as regular optimization and MD.

5.4.1 Energy calculation

class `pydlpoly.`**`pydlpoly`**(*name*, *instances=None*)

`calc_energy`(*force=True*)
compute the current energy, by default also the force is computed Note that in dlpoly the force is always computed (even with force=False)

`calc_energy_force`()
Computes energy with calc_energy and returns also current forces

Returns

- energy (energy in kcal/mol)
- fxyz [natoms, 3]

5.4.2 Getting and setting various things

The following methods allow to get and set various properties like atom positions and velocities (of the complete system or molecular subset) or the cell paramters.

class `pydlpoly.`**`pydlpoly`**(*name*, *instances=None*)

get_atomtypes()
get a list of the atomtypes

get_bcond()
get the boundary conditions as an integer

- 0 : non-periodic
- 1 : cubic
- 2 : orthormobic
- 3 : triclinic
- 6 : 2D periodic

for higher numbers see DL_Poly documentation

get_cell()
get curernt cell vectors as a numpy array [3,3]

get_charges()
get the atomic charges as numpy array [natoms]

get_elements()
get a list of all elements derived from the atomtypes

get_frac()
get the current coordinates as fractional coordinates

get_masses()
get atomic masses

get_natoms()
get the systems number of atoms

get_stress()
get the stress tensor as a numpy array [3,3] in kcal/mol/A^2

get_subset_masses(*aind*)
get masses for atoms indexed in list aind

get_subset_vel(*aind*)
returns velocities (internal units Angstrom per picosecond) as a numpy arrays

get_subset_xyz(*aind*)
get atom positions for a subset as a numpy array :Parameters:

- aind : list of atom indices (starting with 0, Python style)

get_temperature()
get the (reference) system temperature

get_tstep()
get the timestep in ps

get_xyz()
get the atomic positions as a numpy array for convenience we remap the linear coordinates in a [N,3] shape

set_atoms_moved()
tells the system that the energy/forces are invalid and need to be recomputed. most methods like pydlpoly.set_xyz do this automatically So it is usually not necessary to call this from the user side

set_cell(*cell*, *cell_only=False*)
set cell from cell vectors

Parameters

- cell [3,3] : numpy array with cell vectors
- cell_only : boolean if true only the cell is changed but the atom postions are untouched. by default the atoms are scaled according to the change of the cell

set_charges(*q*)
set the atomic charges

set_frac(*frac*)
set coordinates from fractional coords

set_subset_vel(*new_vel*, *aind*)
set subset velocites from numpy array (see set_subste_xyz)

set_subset_xyz(*new_xyz*, *aind*)
same as `set_xyz` for a subset :Parameters:

•new_xyz [len(aind), 3] : numpy array with cartesian coordinates in Angstrom

•aind : list of atom indices (starting with 0, Python style)

set_vel(*new_vel*)
set velocities from numpy array

set_xyz(*new_xyz*)
set all atom positions from numpy array :Parameters:

•new_xyz [natoms, 3] : numpy array with cartesian coordinates in Angstrom

5.4.3 Temperatures and Thermostating

class `pydlpoly.`**pydlpoly**(*name*, *instances=None*)

get_degfree()
get degrees of dreedom from Fortran

set_degfree(*dof*)
set degrees of freedom and sigma (for thermostat) in fortran

set_thermostat_sigma(*degfree*)
resets the sigma value used in the thermostat to determine the reference kin energy

5.4.4 Additional systems or potential terms

An extra term is an additional energy expression for the molecular system which is evaluated after the regular molecular mechanics energy evaluation. An extra system also contains additional degrees of freedom and the extra system must be a class with a defined interface containing methods to propagate the system with a velocity verlet propagator etc. (see special docu on extra systems).

class `pydlpoly.`**pydlpoly**(*name*, *instances=None*)

add_extra_system(*system*)
add an extra system (class instance with a defined API, see docu for extra systems) There can be multiple extra systems. Extra systems have there own degrees of freedom which are propagated (which is the difference to an extra_term)

Parameters

- system : instance of a class with the extra system API

Note Energy and forces of the extra systen are calculated after the regular pydlpoly energies are computed

set_extra_term(*efunc*)
add an additional energy term (called AFTER standard pydlpoly terms)

Parameters

- efunc : callable object (without paramters) computes an extra energy and forces

Note There can be only one extra term

5.4.5 Writing structures

class pydlpoly.**pydlpoly**(*name*, *instances=None*)

write_tinker_xyz(*fname*, *fullcell=None*, *mode='w'*, *moldenr=False*)
if fullcell is not set we write the cell parameters (a,b,c,alpha,beta,gamma) and rotate the system properly
if fullcell is true we write the complete cell vectors (9 values) to the first line

Parameters

- moldenr : replaces atom type strings with integers to ensure readability by molden when set to True

write_xyz(*fname*)
writing out a simple xyz file for DEBUG purposes

Todo

Need more details here. Complete the docstrings including Paramters and Returns of all methods to be reasonably called from users in the pydlpoly.py file

---

CHAPTER

SIX

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

Symbols

A

C

G

L

M

P

S

W