RadonPy tutorial

Yoshihiro Hayashi
The Institute of Statistical Mathematics
yhayashi@ism.ac.jp

31th Mar 2022

Python library for automated polymer MD

RadonPy

Automated pipeline of MD calculation for polymer properties seamless integrated with RDKit

Pre-process

Polymer chain generator
- Homopolymer
- Alternating copolymer
- Random copolymer
- Block copolymer

Unit cell generator
- Amorphous
  - Single component
  - Mixture
- Oriented (Nematic-like)
  - Single component
  - Mixture
- Crystalline

Force field assignment
- GAFF
- GAFF2

Atomic charge calculation
- Gasteiger
- RESP (using QM solver)
- ESP (using QM solver)

Input file generator
- NVE, NVT, NPT
- Annealing
- Thermal conduction
- Uni-axial extension

Simulation

MD solver
- LAMMPS

QM solver
- Psi4

Post-process

Property calculation
- Density
- Cp, Cv
- Bulk modulus
- Thermal expansion
- Linear expansion
- Static dielectric const.
- Thermal conductivity
- Order parameter
- Young’s modulus, etc

Data science

SMILES transformation
- Cyclic polymer
- Linear polymer

Descriptor calculation
- Force field descriptor
  - Summary statistics
  - Kernel mean

Recommended system requirement

- Python 3.7, 3.8, 3.9
- RDKit >= 2020.03
- Psi4 >= 1.5
- resp
- dftd3
- mdtraj >= 1.9
- matplotlib
- scipy
- pandas
- LAMMPS >= 3Mar20

Installation

1. Download of miniforge
https://github.com/conda-forge/miniforge

2. Installation of miniforge on Linux system

```
bash ./Miniforge3-Linux-x86_64.sh
```

3. Create conda environment (Python 3.9)

```
conda create -n radonpy python=3.9
conda activate radonpy
```

4. Installation of requirement packages by conda

```
conda install -c psi4 -c conda-forge rdkit psi4 resp scipy mdtraj matplotlib pandas
```

5. Installation of LAMMPS by conda

```
conda install -c conda-forge lammps
```

or manually build from source of LAMMPS official site.
In this case, the environment variable must be set:

```
export LAMMPS_EXEC=<path-to-LAMMPS-binary>
```

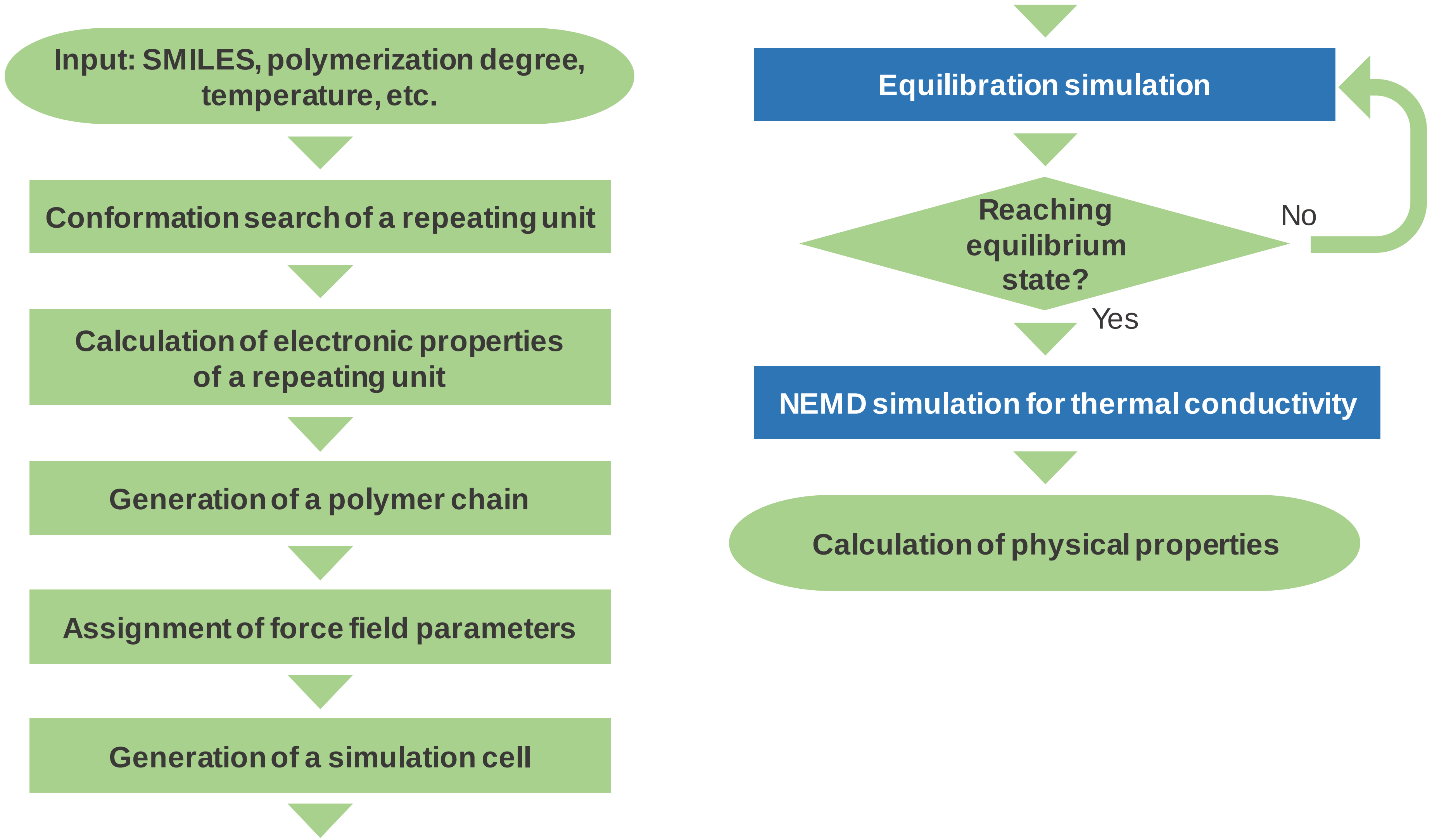
6. Installation of RadonPy

```
git clone -b main https://github.com/RadonPy/RadonPy.git
export PYTHONPATH=<Path-to-RadonPy>:$PYTHONPATH
```

Module structure

- radonpy
  - core
    - poly — Polymer module: generation of polymer chain, generation of simulation cell
    - utils — Utilities: transformation to Mol object from SMILES, output for PDB and pickle file
    - calc — Internal function of calculations
    - const — Physical constants, conversion factors
  - ff
    - gaff — Assignment of GAFF
    - gaff2 — Assignment of GAFF2
    - gaff2_mod — Assignment of GAFF2 using modified parameters
    - descriptor — Force field descriptor
  - sim
    - qm — QM calculation
    - md — Setting of MD calculation condition
    - lammps — LAMMPS execution, analysis of MD results
    - psi4_wrapper — Wrapper module of Psi4
    - preset
      - eq — Preset: equilibration
      - tc — Preset: thermal conductive NEMD

Example of workflow



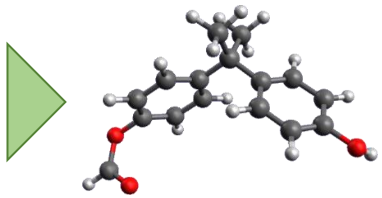
Repeating unit generation from SMILES

radonpy.core.utils.mol_from_smiles

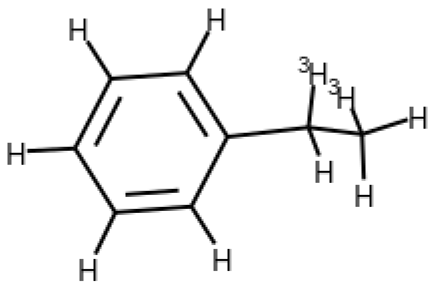
```
from radonpy.core import utils

smiles = '*CC(*)c1ccccc1'
mol = utils.mol_from_smiles(smiles)
```

SMILES **Monomer**

Oc1ccc(cc1)C(c1ccc(cc1)OC(=O))(C)C



- For a given SMILES string of a polymer repeating unit, 3D atomic were generated using the ETKDG method.
- The input SMILES string has two asterisk symbols for representing two attachment points of the repeating unit.
- The two attachment points were capped with tritium atoms.
- For unidentified *cis*/*trans* isomer, The *trans* isomer is generated in default.
- For unidentified *R*/*S* chiral isomer, The *S* isomer is generated in default.



Example of a repeating unit of polystyrene

RDKit Mol object

RDKit Mol object contains information of atoms, bonds, atomic coordinates, etc.
RadonPy extends the RDKit Mol object for polymer modeling and MD calculation.

Atom object

```
Mol.GetAtoms()
atom = Mol.GetAtomWithIdx(idx)
```

Returns a sequence containing all of the molecule's Atoms.
Returns a particular Atom.

```
atom.GetIdx()
atom.GetAtomicNum()
atom.GetSymbol()
atom.GetMass()
```

Returns the atom's index
Returns the atomic number.
Returns the atomic symbol
Returns the atomic mass.

Bond object

```
Mol.GetBonds()
bond = Mol.GetBondWithIdx(idx)
```

Returns a sequence containing all of the molecule's Bonds.
Returns a particular Bond.

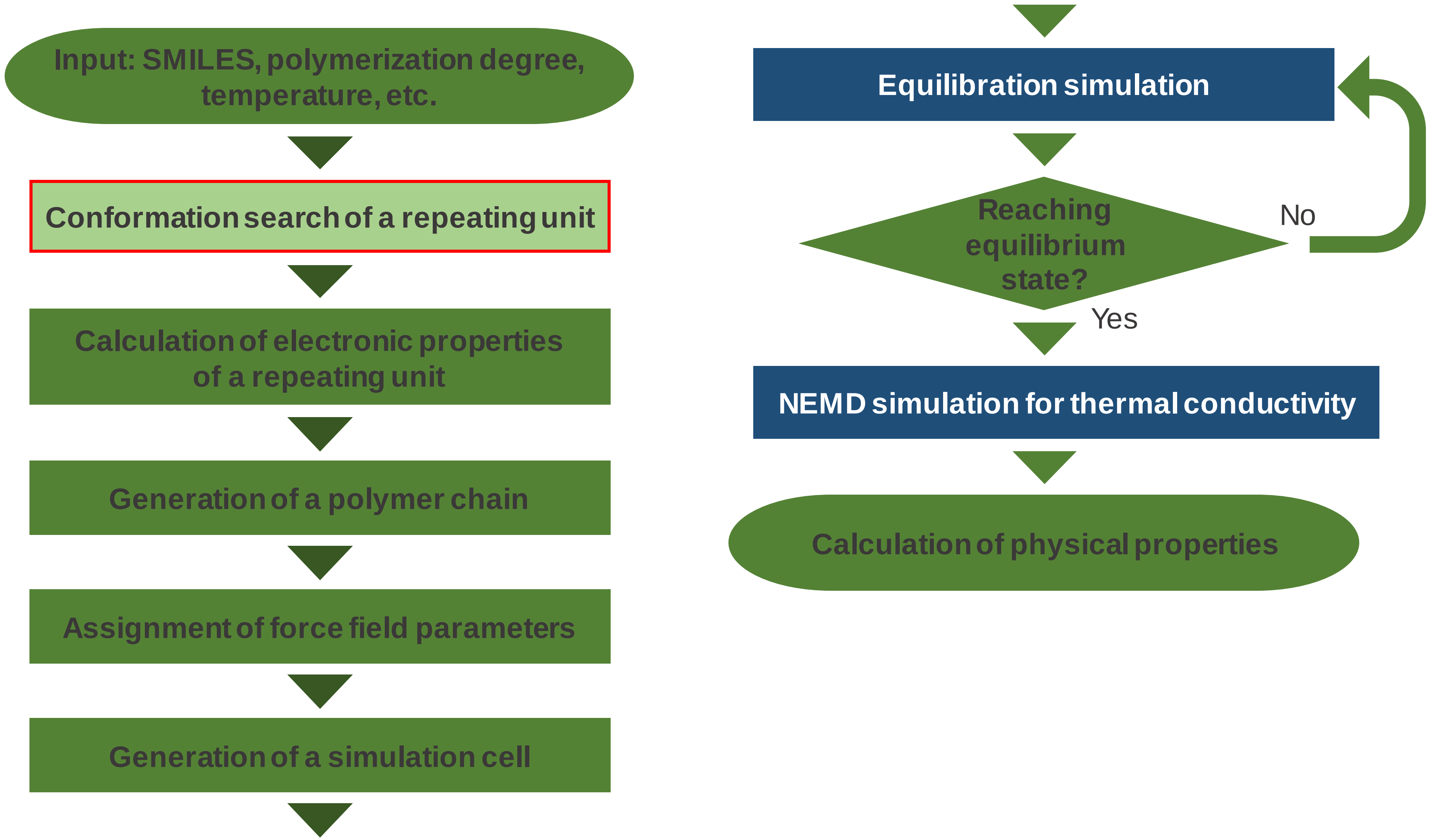
```
bond.GetIdx()
bond.GetBondTypeAsDouble()
bond.IsInRing()
bond.GetBeginAtom()
bond.GetEndAtom()
```

Returns the bond's index
Returns the type of the bond as a double (i.e. 1.0 for SINGLE, 1.5 for AROMATIC, 2.0 for DOUBLE)
Returns whether or not the bond is in a ring of any size.
Returns the bond's first atom.
Returns the bond's second atom.

Reference sites
https://www.rdkit.org/docs/

(Japanese)
https://www.rdkit.org/docs_jp/
https://future-chem.com/rdkit-mol/

Conformation search



Conformation search

```
from radonpy.core import utils
from radonpy.sim import qm
from radonpy.ff.gaff2_mod import GAFF2_mod

ff = GAFF2_mod()    # Instance of GAFF2_mod force field
mol = utils.mol_from_smiles(smiles)   # Generation of mol object
mol, energy = qm.conformation_search(mol, ff=ff, work_dir=work_dir, psi4omp=omp_psi4, mpi=mpi, omp=omp, gpu=gpu, log_name='monomer1')
```

qm.conformation_search automatically executes following procedure for a given Mol object of a polymer repeating unit

1. **[Conformation generation]** 3D atomic coordinates of up to **<u>1,000</u>** different conformations were generated using the ETKDG version 2 method implemented in RDKit.

2. **[MM optimization]** The potential energy of each conformation of a repeating unit was evaluated using the molecular mechanics calculation with GAFF2_mod after the geometry optimization.

3. **[Clustering]** The optimized conformers were clustered by performing the Butina clustering based on the torsion fingerprint deviation.

4. **[DFT optimization]** The most stable **<u>4</u>** conformations were further optimized by performing DFT calculations with the ωB97M-D3BJ/6-31G(d,p).

5. **[Most stable conformation]** Conformation IDs in the returned Mol object are sorted by the DFT energy. The most stable conformation is set to 0 of conformation ID.

radonpy.sim.qm.conformation_search

```
def conformation_search(mol, ff=None, nconf=1000, dft_nconf=4, etkdg_ver=2, rmsthresh=0.5, tfdthresh=0.02, clustering='TFD',
                        opt_method='wb97m-d3bj', opt_basis='6-31G(d,p)', opt_basis_gen={'Br': '6-31G(d,p)', 'I': 'lanl2dz'},
                        geom_iter=50, geom_conv='QCHEM', geom_algorithm='RFO', log_name='mol', solver='lammps',
                        solver_path=None, work_dir=None, tmp_dir=None, etkdg_omp=-1, psi4_omp=-1, psi4_mp=1,
                        omp=1, mpi=-1, gpu=0, mm_mp=1, memory=1000, **kwargs):
```

mol: Input of RDKit Mol object

Options

ff: Force field object for MM optimization

nconf: Maximum number of generated conformations by ETKDG [int]

dft_nconf: Maximum number of conformations to be geometry optimized by DFT calculations [int]

etkdg_ver: Version of ETKDG method [int]

rmsthresh: Threshold of RMS for clustering in conformation generation with ETKDG [float]

tfdthresh: Threshold of torsion fingerprint deviation in conformation clustering [float]

opt_method: Calculation method of the DFT optimization [str]

opt_basis: Basis set of the DFT optimization [str]

geom_iter: Maximum number of iteration in the DFT optimization [int]

geom_conv: Conversion criterion in the DFT optimization [str]

geom_algorithm: Algorithm in the DFT optimization [str]

log_name: Prefix of the file name of Psi4 log in the DFT optimization [str]

work_dir: Working directory [str]

psi4_omp: Parallel number of OpenMP in DFT optimization by Psi4 [int]

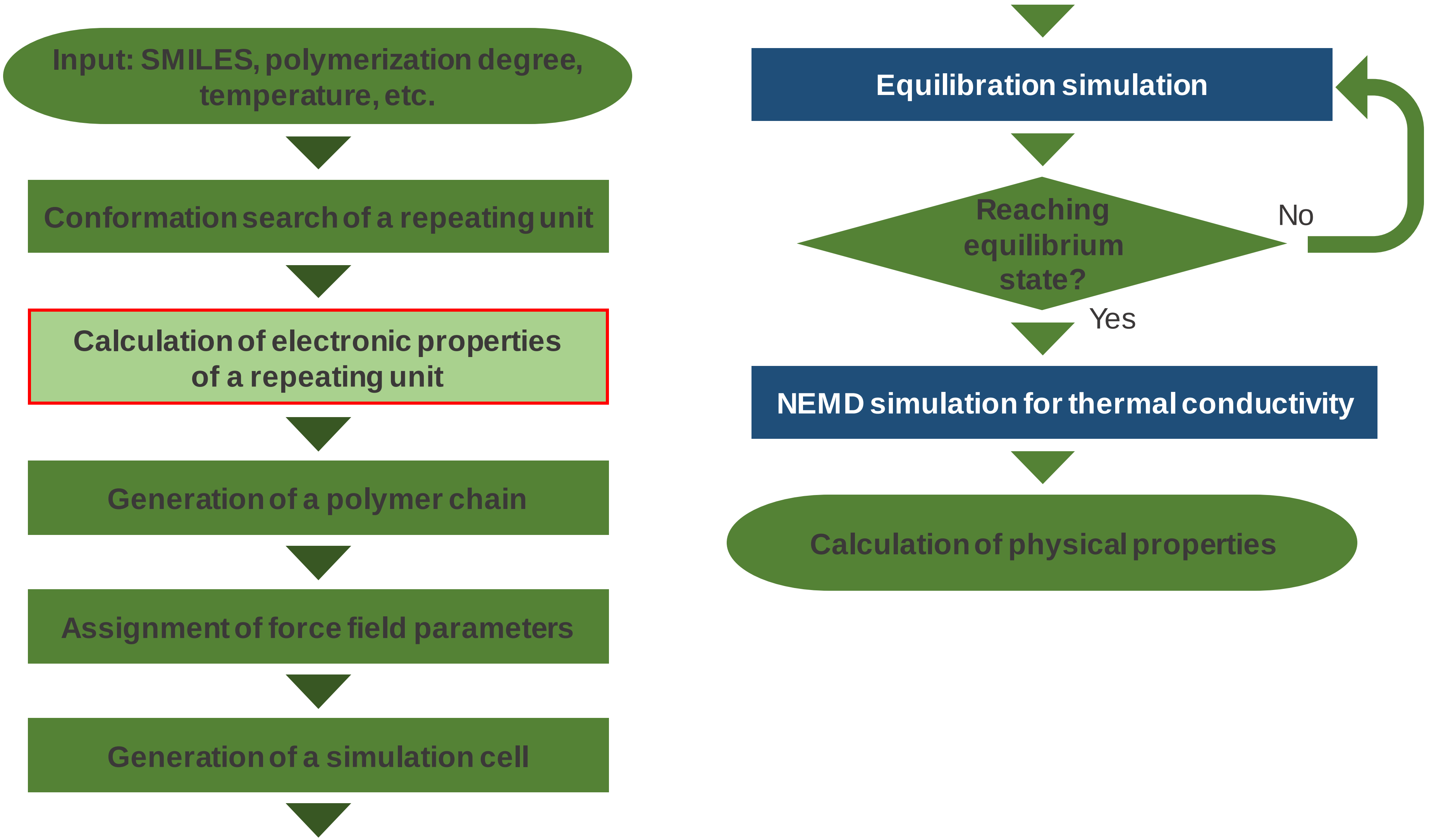
omp: Parallel number of OpenMP in MM calculation by LAMMPS [int]

mpi: Process number of MPI in MM calculation by LAMMPS [int]

Returns

mol: RDKit Mol object, energy: Energy values of DFT and MM

Electronic property calculation



Electronic property calculation

```
from radonpy.sim import qm

qm.assign_charges(mol, charge='RESP', work_dir=work_dir, tmp_dir=tmp_dir, opt=False, omp=omp_psi4, memory=mem_psi4)
qm_data = qm.sp_prop(mol, opt=False, work_dir=work_dir, tmp_dir=tmp_dir, omp=omp_psi4, memory=mem_psi4)
polar_data = qm.polarizability(mol, opt=False, work_dir=work_dir, tmp_dir=tmp_dir, omp=conf_psi4_omp, memory=mem_psi4)
```

radonpy.sim.qm

assign_charges: Calculation and assignment atomic charge of the input Mol object

sp_prop: Calculation of total energy, HOMO, LUMO, dipole moment of the input Mol Object

polarizability: Calculation of polarizability tensor of the input Mol object

Notice: The electronic property calculation should perform for a repeating unit not for polymer chain because this process is very time consuming.

radonpy.sim.qm.assign_charges

Calculation and assignment of atomic charge

```
def assign_charges(mol, charge='RESP', confId=0, opt=True, work_dir=None, tmp_dir=None, log_name='charge',
                   opt_method='wb97m-d3bj', opt_basis='6-31G(d,p)', geom_iter=50, geom_conv='QCHEM', geom_algorithm='RFO',
                   charge_method='HF', charge_basis='6-31G(d)', charge_basis_gen={'Br':'6-31G(d)', 'I': 'lanl2dz'}, **kwargs):
```

mol: Input of RDKit Mol object

Options

charge: Charge model (RESP, ESP, gasteiger, Mulliken, Lowdin) [str]
confId: Conformation ID of the RDKit mol object [int]
(confId=0 is the most stable conformation after conformation_search)
opt: With (True) /Without (False) geometry optimization by DFT calculation before the charge calculation [boolean]
work_dir: Working directory [str]
omp: Parallel number of OpenMP in Psi4 [int]
memory: Amount of memory usage (MB) in Psi4 [int]
log_name: Prefix of the file name of Psi4 log [str]
opt_method: Calculation method of the geometry optimization [str]
opt_basis: Basis set of the geometry optimization [str]
geom_iter: Maximum number of iteration in the geometry optimization [int]
geom_conv: Conversion criterion in the geometry optimization [str]
geom_algorithm: Algorithm in the geometry optimization [str]
charge_method: Calculation method of the charge calculation [str]
(The default method is HF because of the GAFF recommendation)
charge_basis: Basis set of the charge calculation [str]
charge_basis_gen: Basis set of the charge calculation for each element [dict]

Returns

Boolean: Success (True) / Fail (False)

radonpy.sim.qm.sp_prop

Calculation of total energy, HOMO level, LUMO level, and dipole moment

```
def sp_prop(mol, confId=0, opt=True, work_dir=None, tmp_dir=None, log_name='sp_prop',
            opt_method='wb97m-d3bj', opt_basis='6-31G(d,p)', opt_basis_gen={'Br': '6-31G(d,p)', 'I': 'lanl2dz'},
            geom_iter=50, geom_conv='QCHEM', geom_algorithm='RFO',
            sp_method='wb97m-d3bj', sp_basis='6-311G(d,p)', sp_basis_gen={'Br': '6-311G(d,p)', 'I': 'lanl2dz'}, **kwargs):
```

mol: Input of RDKit Mol object

Options

confId: Conformation ID of the RDKit mol object [int]
opt: With (True) /Without (False) geometry optimization by DFT calculation [boolean]
work_dir: Working directory [str]
omp: Parallel number of OpenMP in Psi4 [int]
memory: Amount of memory usage (MB) in Psi4 [int]
log_name: Prefix of the file name of Psi4 log [str]
opt_method: Calculation method of the geometry optimization [str]
opt_basis: Basis set of the geometry optimization [str]
opt_basis_gen: Basis set of the geometry optimization (for each element) [str]
geom_iter: Maximum number of iteration in the geometry optimization [int]
geom_conv: Conversion criterion in the geometry optimization [str]
geom_algorithm: Algorithm in the geometry optimization [str]
sp_method: Calculation method of the single-point calculation [str]
sp_basis: Basis set of the single-point calculation [str]
sp_basis_gen: Basis set of the geometry optimization (for each element) [dict]

Returns

dict type
- qm_total_energy: Total enelgy (float, kJ/mol)
- qm_homo: HOMO (float, eV)
- qm_lumo: LUMO (float, eV)
- qm_dipole_x, qm_dipole_y, qm_dipole_z Dipole moment (float, Debye)

radonpy.sim.qm.polarizability

Calculation of polarizability tensor

```
def polarizability(mol, confId=0, opt=True, work_dir=None, tmp_dir=None, log_name='polarizability', mp=1,
                   opt_method='wb97m-d3bj', opt_basis='6-31G(d,p)', opt_basis_gen={'Br': '6-31G(d,p)', 'I': 'lanl2dz'},
                   geom_iter=50, geom_conv='QCHEM', geom_algorithm='RFO',
                   polar_method='wb97m-d3bj', polar_basis='6-311+G(2d,p)', polar_basis_gen={'Br': '6-311G(d,p)', 'I': 'lanl2dz'}, **kwargs):
```

mol: Input of RDKit Mol object

Options

confId: Conformation ID of the RDKit mol object [int]
opt: With (True) /Without (False) geometry optimization by DFT calculation) [boolean]
work_dir: Working directory [str]
omp: Parallel number of OpenMP in Psi4 [int]
memory: Amount of memory usage (MB) in Psi4 [int]
log_name: Prefix of the file name of Psi4 log [str]
opt_method: Calculation method of the geometry optimization [str]
opt_basis: Basis set of the geometry optimization [str]
opt_basis_gen: Basis set of the geometry optimization (for each element) [str]
geom_iter: Maximum number of iteration in the geometry optimization [int]
geom_conv: Conversion criterion in the geometry optimization [str]
geom_algorithm: Algorithm in the geometry optimization [str]
polar_method: Calculation method of the polarizability calculation [str]
polar_basis: Basis set of the polarizability calculation [str]
polar_basis_gen: Basis set of the polarizability calculation (for each element) [dict]

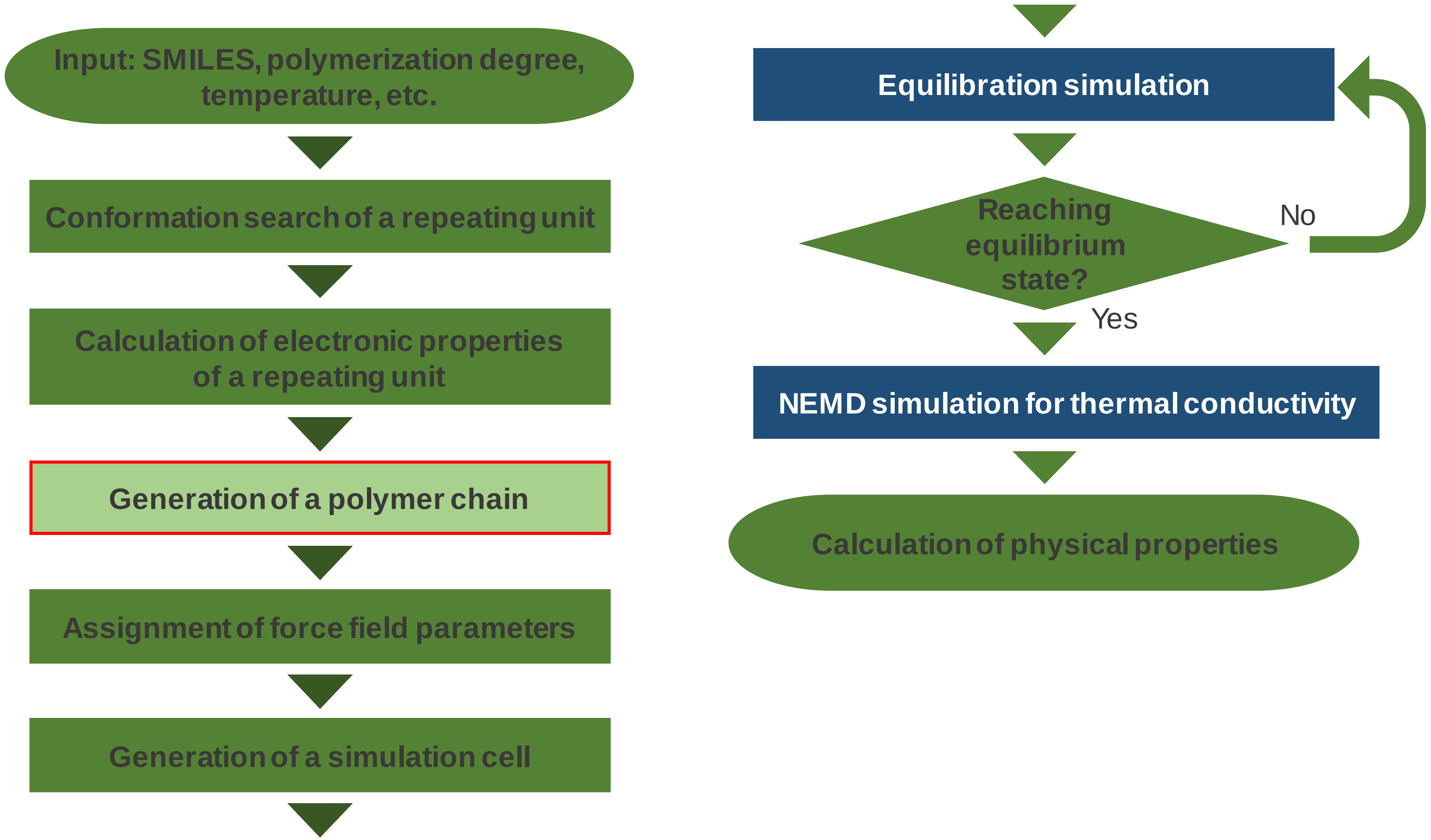
Returns

dict type

qm_polarizability (float, $Å^3$)

qm_polarizability_xx, qm_polarizability_yy, qm_polarizability_zz,
qm_polarizability_xy, qm_polarizability_xz, qm_polarizability_yz (float, $Å^3$)

Generation of a polymer chain generation



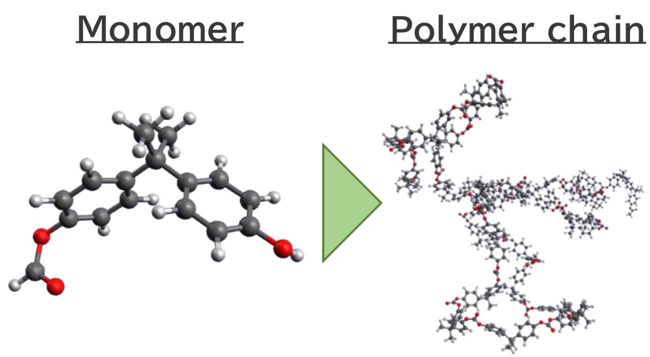
Polymer chain generation: homopolymer

```
from radonpy.core import poly, utils

smiles = '*CC(*)c1ccccc1'
mol = utils.mol_from_smiles(smiles)
ter = utils.mol_from_smiles('*C')

n = poly.calc_n_from_num_atoms(mol, 1000, terminal1=ter)

homopoly = poly.polymerize_rw(mol, n)
homopoly = poly.terminate_rw(homopoly, ter)
```

poly.calc_n_from_num_atoms(mol, natom, terminal1)
Calculate the degree of polymerization to achieve the specified number of atoms (*natom*)

poly.polymerize_rw(mol, n)
Generation of a polymer chain with self-avoiding random walk

poly.terminate_rw(mol, ter)
Termination of a polymer chain



radonpy.core.poly.polymerize_rw

Generation of a homopolymer chain by self-avoiding random walk

```
def polymerize_rw(mol, n, headhead=False, confId=0, tacticity='atactic', atac_ratio=0.5, tac_array=None,
                  retry=10, retry_step=100, dist_min=0.7, opt='rdkit', ff=None, work_dir=None, omp=1, mpi=1, gpu=0):
```

mol: RDKit Mol object of repeating unit
n: Degree of polymerization [int]

Options

headhead: Connection type of repeating units, head-to-head (True) / head-to-tail (False) [boolean]
confId: Conformation ID of the RDKit mol object [int]
tacticity: Select tacticity (atactic, isotactic, or syndiotactic) [str]
atac_ratio: R/S ratio of atactic polymer [float]
tac_array: Specify the array of R/S isomer [list of boolean]
retry: Maximum number of retry for this function when generating unsuitable structure [int]
retry_step: Maximum number of retry for a random-walk step when generating unsuitable structure [int]
dist_min: Threshold of minimum atom-atom distance (angstrom) [float]
opt: Optimization method in polymer chain growing [str]

Returns

RDKit Mol object

Polymer chain generation: copolymer

Random copolymer

```
n = poly.calc_n_from_num_atoms([mol, mol2], 1000, ratio=[0.5, 0.5], terminal1=ter)

rcopoly = poly.random_copolymerize_rw([mol,  mol2], n, ratio=[0.5, 0.5])
rcopoly = poly.terminate_rw(rcopoly,  ter)
```

poly.random_copolymerize_rw(mols, n, ratio)

mols: List of RDKit Mol object of repeating unit

n: Degree of polymerization [int]

<u>Options</u>

ratio: Component ratio [list of float]

Alternating copolymer

```
acopoly = poly.copolymerize_rw([mol,  mol2], 20)
acopoly = poly.terminate_rw(acopoly,  ter)
```

poly.copolymerize_rw(mols, n)

mols: List of RDKit Mol object of repeating unit

n: Degree of polymerization [int]

Block copolymer

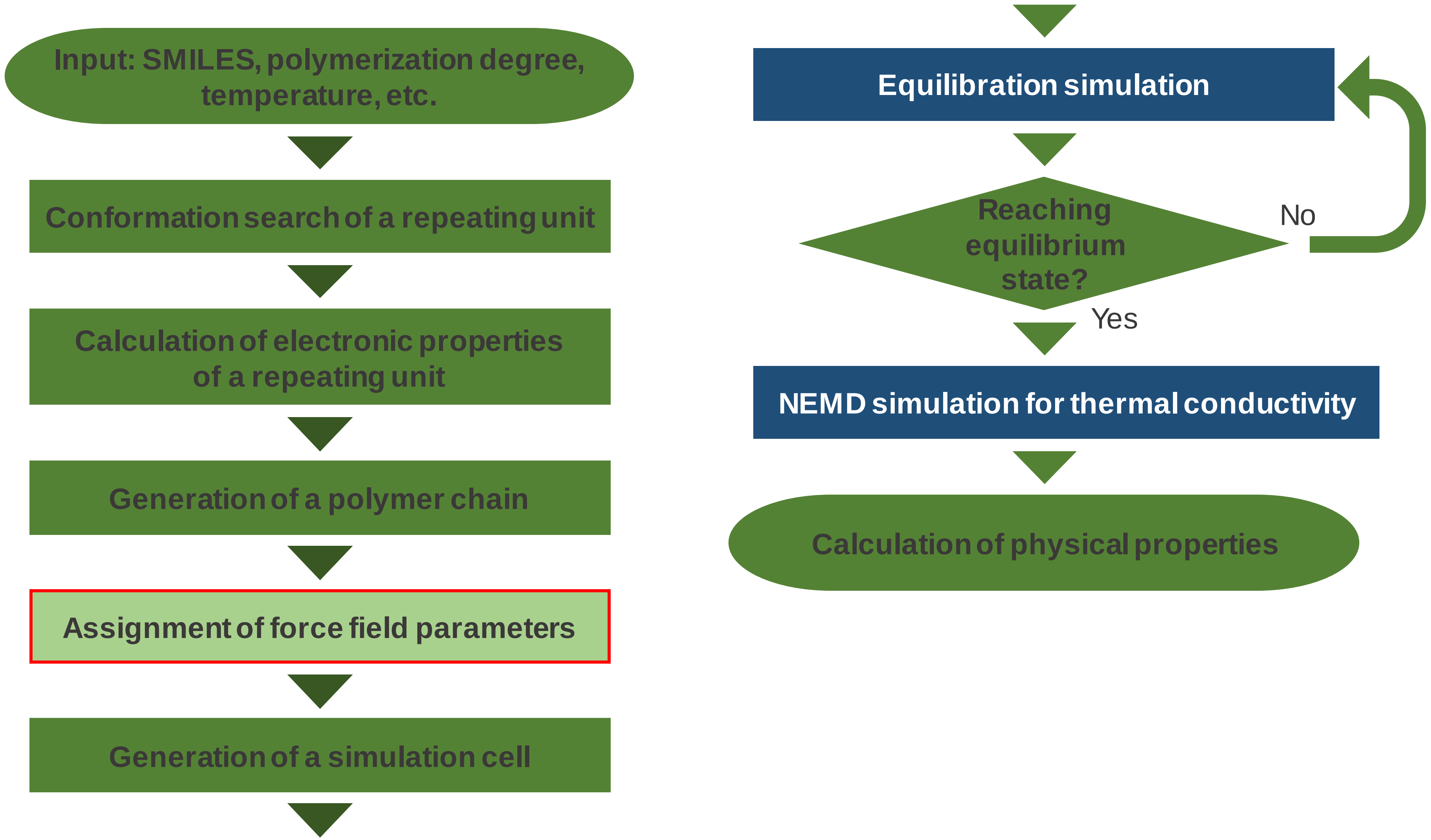
```
bcopoly = poly.block_copolymerize_rw([mol,  mol2], [20, 20])
bcopoly = poly.terminate_rw(bcopoly,  ter)
```

poly.block_copolymerize_rw(mols, n)

mols: List of RDKit Mol object of repeating unit

n: List of degree of polymerization [list of int]

Assignment of force field parameters



Force field assignment

GAFF2 assignment

```
from radonpy.ff.gaff2 import GAFF2

ff = GAFF2()
result = ff.ff_assign(homopoly)
```

- Available elements : H, C, N, O, F, P, S, Cl, Br, I
- Original parameter set of GAFF2
- Return value: True (assignment success) / False (assignment fail)

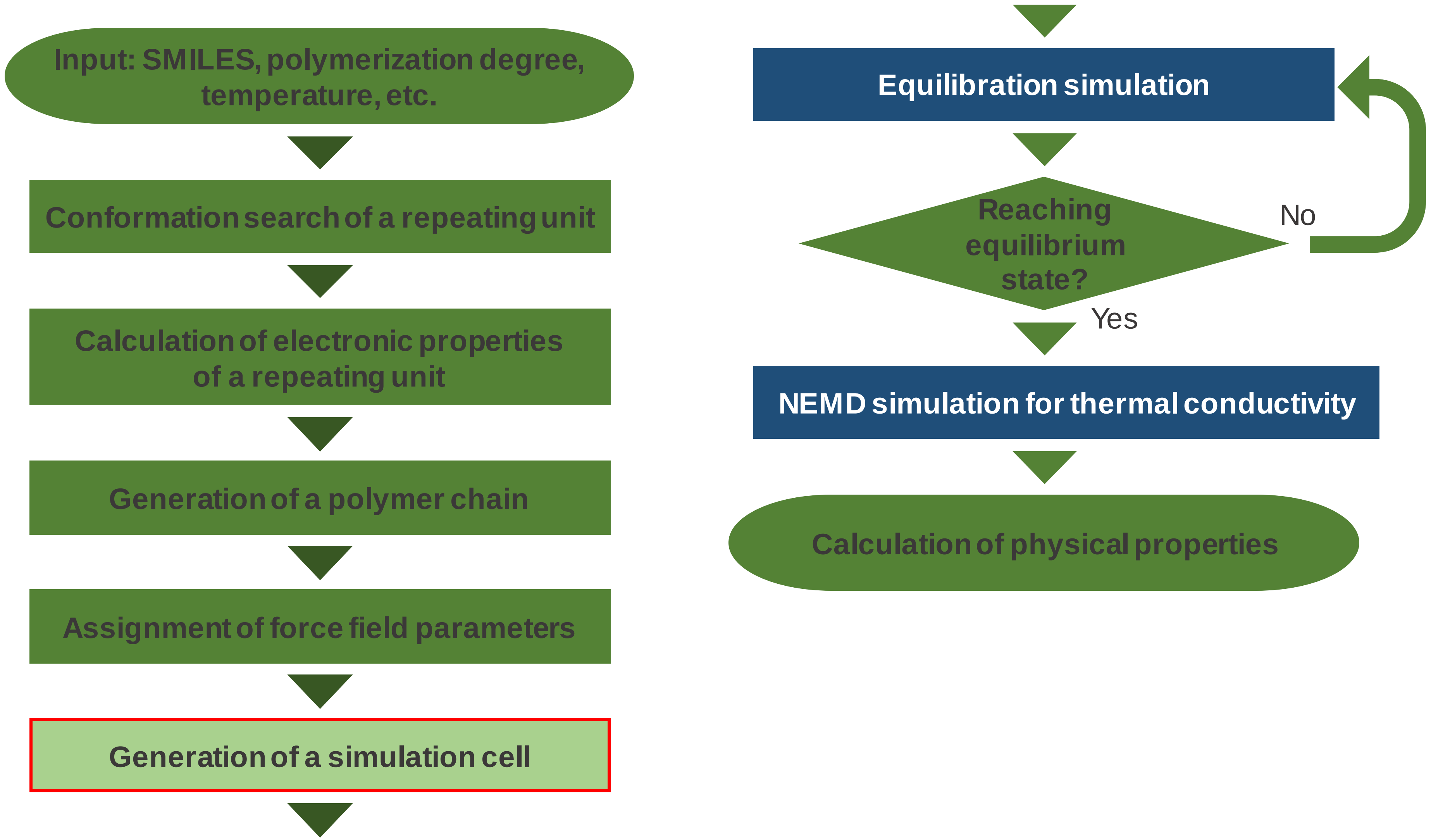
Modified GAFF2 assignment

```
from radonpy.ff.gaff2 import GAFF2_mod

ff = GAFF2_mod()
result = ff.ff_assign(homopoly)
```

- Available elements : H, C, N, O, F, P, S, Cl, Br, I
- Modified parameters are used for fluorocarbon.
  - J. Trag, D, Zahn, *J. Mol. Model.* (2019) *25*, 39
- Improving density evaluation of fluorocarbon polymers

Notice: The force field assignment should perform for a polymer chain not for simulation cell because this process is time consuming.

Generation of a simulation cell

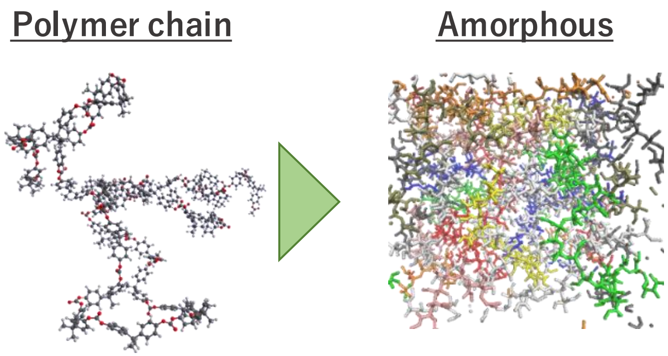


Amorphous cell generation

Generation of simulation cell of amorphous polymers (single component)

```
ac = poly.amorphous_cell(homopoly, 10, density=0.05)
```

The cell was constructed by randomly arranging and rotating
10 polymer chains such that they did not overlap with each other.




```
def amorphous_cell(mol, n, cell=None, density=0.03, retry=10, retry_step=100, threshold=2.0, dec_rate=0.8)
```

mol: RDKit Mol object of repeating unit
n: Number of replication of Mol object [int]

Options
cell: Adding to existing cell [Mol object]
density: Density of generating cell [float]
retry: Maximum number of retry for this function when generating unsuitable structure [int]
retry_step: Maximum number of retry for a randomly arranging step when generating unsuitable structure [int]
threshold: Threshold of minimum atom-atom distance (angstrom) [float]

Returns
RDKit Mol object

Amorphous cell generation (mixture)

Generation of simulation cell of amorphous polymers (multi component)

```
ac = poly.amorphous_mixture_cell([poly1, poly2], [5, 5], density=0.1)
```

The cell was constructed by randomly arranging and rotating five poly1 and five poly2 such that they did not overlap with each other.

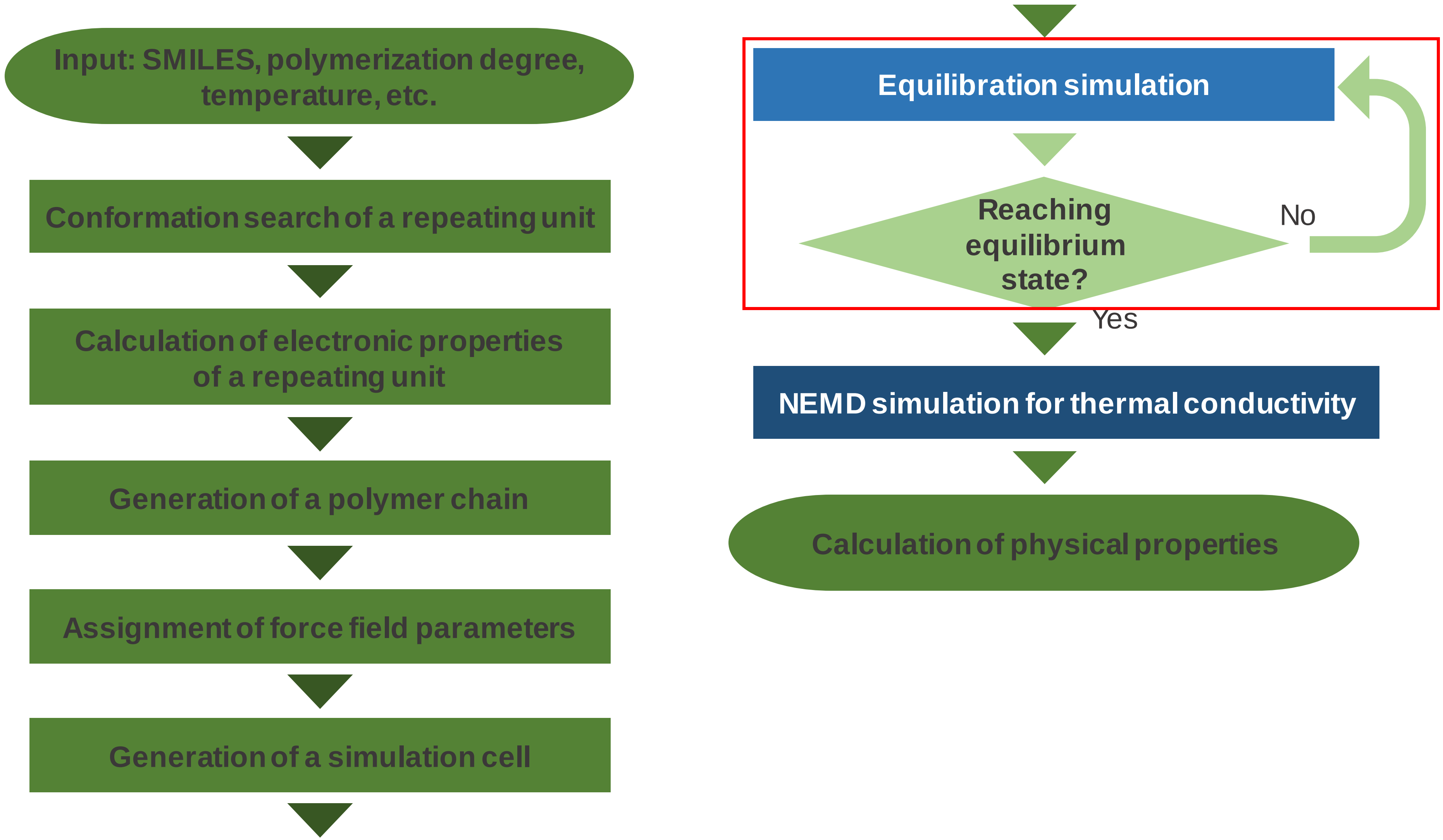
```
def amorphous_mixture_cell(mols, n, cell=None, density=0.03, retry=10, retry_step=100, threshold=2.0, dec_rate=0.8)
```

mols: List of RDKit Mol object of repeating unit
n: List of number of replication of Mol object [list of int]

Options
cell: Adding to existing cell [Mol object]
density: Density of generating cell [float]
retry: Maximum number of retry for this function when generating unsuitable structure [int]
retry_step: Maximum number of retry for a randomly arranging step when generating unsuitable structure [int]
threshold: Threshold of minimum atom-atom distance (angstrom) [float]

Returns
RDKit Mol object

Equilibration simulation



radonpy.sim.preset.eq.EQ21step

Preset of equilibration using the 21-steps compression/decompression protocol

```
from radonpy.sim.preset import eq

eqmd = eq.EQ21step(ac, work_dir=work_dir)
ac = eqmd.exec(temp=300, press=1.0, mpi=mpi, omp=omp, gpu=gpu)
```

eq.EQ21step.exec performs following 3 steps:

1. Packing simulation
Increase the density of the amorphous polymers to an appropriate value for subsequent calculations

2. The 21-steps compression/decompression protocol
[G. S. Larsen, P. Lin, K. E. Hart, and C. M. Colina, *Macromolecules* **44**, 6944–6951 (2011).]
A temperature rise to 600 K and a drop to 300 K were repeated for approximately 1.5 ns while the system was compressed to 50,000 atm and then decompressed to 1 atm by combining the NVT and NpT simulations

3. NPT equilibration
Run for 5 ns at 300 K and 1 atm

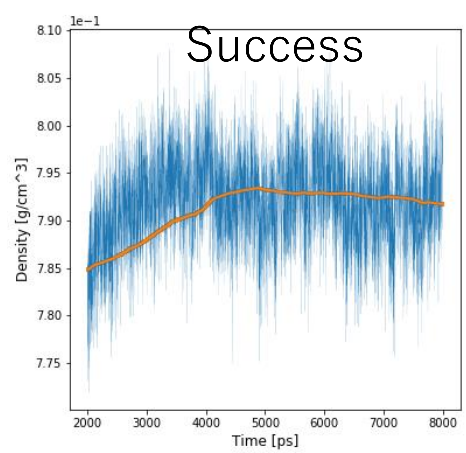
Physical property calculations from EMD

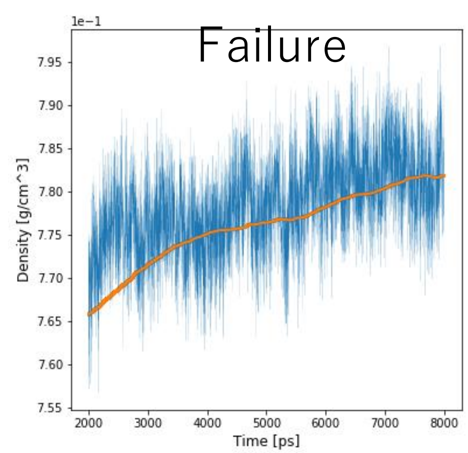
```
from radonpy.sim.preset import eq

eqmd = eq.EQ21step(ac, work_dir=work_dir)
ac = eqmd.exec(temp=300, press=1.0, mpi=mpi, omp=omp, gpu=gpu)

analy = eqmd.analyze()
prop_data = analy.get_all_prop(temp=300, press=1.0, save=True)
result = analy.check_eq()
```

eqmd.analyze()
Return analysis object

analy.get_all_prop(temp=temp, press=press, save=True)
Calculation of physical property from equilibration MD
<u>Return</u>
dict type (containing calculated physical properties)

analy.check_eq()
Determines the completion of the relaxation of the equilibration calculation based on the convergence status for energies, density, and Rg.
This function must be executed after get_all_prop.
<u>Return</u>
Boolean: True (reached equilibrium state)
False (did not reach equilibrium state)





Additional EMD

Preset of additional equilibration

```
from radonpy.sim.preset import eq

eqmd = eq.Additional(ac, work_dir=work_dir)
ac = eqmd.exec(temp=300, press=1.0, mpi=mpi, omp=omp, gpu=gpu)

analy = eqmd.analyze()
prop_data = analy.get_all_prop(temp=300, press=1.0, save=True)
result = analy.check_eq()
```

eq.Additional.exec

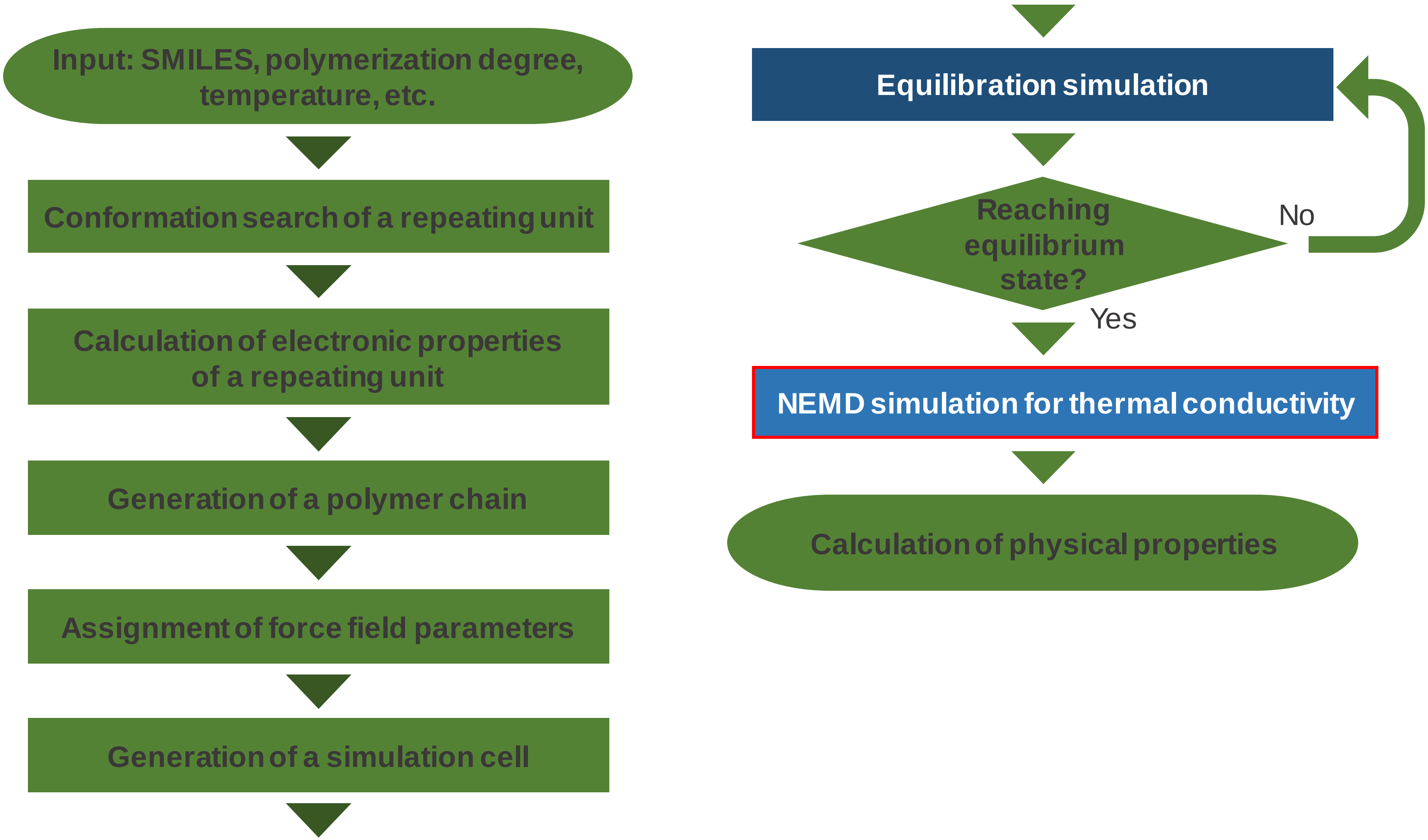
NpT simulations were run for 5 ns at 300 K and 1 atm.
Interface is the same as EQ21step.

Sample code of EMD

```
from radonpy.sim.preset import eq

# Equilibration MD
eqmd = eq.EQ21step(ac, work_dir=work_dir)
ac = eqmd.exec(temp=temp, press=press, mpi=mpi, omp=omp, gpu=gpu)

analy = eqmd.analyze()
prop_data = analy.get_all_prop(temp=temp, press=press, save=True)
result = analy.check_eq()

# Additional equilibration MD
for i in range(4):
    if result: break
    eqmd = eq.Additional(ac, work_dir=work_dir)
    ac = eqmd.exec(temp=temp, press=press, mpi=mpi, omp=omp, gpu=gpu)
    analy = eqmd.analyze()
    prop_data = analy.get_all_prop(temp=temp, press=press, save=True)
    result = analy.check_eq()
```

- Packing simulation and the 21-steps equilibration are performed.
- After the 21-steps equilibration, *NpT* simulations were run for more than 5 ns at 300 K and 1 atm until equilibrium was achieved.
- The achievement of the equilibrium was checked each 5 ns after the 21-steps equilibration.

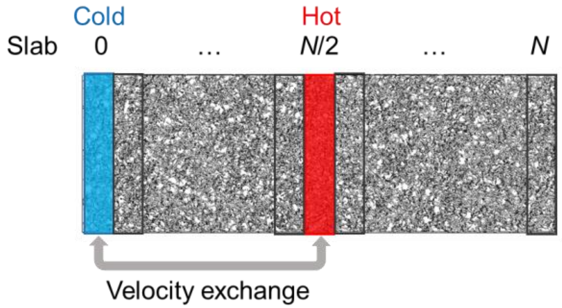
NEMD simulation for thermal conductivity



NEMD simulation of thermal conductivity

Preset of thermal conductive NEMD

```
from radonpy.sim.preset import tc

nemd = tc.NEMD_MP(mol, work_dir=work_dir, axis='x')
mol = nemd.exec(decomp=True, temp=300, mpi=mpi, omp=omp, gpu=gpu)

nemd_analy = nemd.analyze()
TC = nemd_analy.calc_tc(decomp=True, save=True)
TCdecomp = nemd_analy.TCdecomp_data
```



The heat flux is generated in the system with temperature gradients induced by exchanging the velocity between the coldest atom in slab N/2 and the hottest atom in slab 0.

tc.NEMD_MP(mol, work_dir=work_dir, axis='x')
Setup of thermal conductive NEMD with Müller-Plathe method.
axis: specifies the axis of heat flux direction

nemd.exec(decomp=True, temp=temp, mpi=10, omp=1, gpu=0)
Execute thermal conductive NEMD
decomp: With (True) / Without (False) decomposition analysis of thermal conductivity

nemd.analyze()
Return analysis object

nemd_analy.calc_tc(decomp=True, save=True)
Calculation of thermal conductivity.
Return
Thermal conductivity (W/mK) [float]

After nemd_analy.calc_tc:
- nemd_analy.TCdecomp_data : Results of decomposition analysis [dict]
- nemd_analy.Tgrad_data['Tgrad_check']: Result of checking the linearity of temperature gradient [boolean]
- nemd_analy.prop_df : Results of thermal conductivity and decomposition analysis [pandas.DataFrame]

Sample code

```
from radonpy.core import utils, poly
from radonpy.ff.gaff2_mod import GAFF2_mod
from radonpy.sim import qm
from radonpy.sim.preset import eq, tc

smiles = '*C(C*)c1ccccc1'
ter_smiles = '*C'
temp = 300
press = 1.0
omp_psi4 = 10
mpi = 10
omp = 1
gpu = 0
mem = 10000
work_dir = './work_dir'
ff = GAFF2_mod()

if __name__ == '__main__':
    # Conformation search
    mol = utils.mol_from_smiles(smiles)
    mol, energy = qm.conformation_search(mol, ff=ff, work_dir=work_dir,
                                         psi4_omp=omp_psi4, mpi=mpi, omp=omp, memory=mem, log_name='monomer1')

    # Electronic propety calculation
    qm.assign_charges(mol, charge='RESP', opt=False, work_dir=work_dir, omp=omp_psi4, memory=mem, log_name='monomer1')
    qm_data = qm.sp_prop(mol, opt=False, work_dir=work_dir, omp=omp_psi4, memory=mem, log_name='monomer1')
    polar_data = qm.polarizability(mol, opt=False, work_dir=work_dir, omp=omp_psi4, memory=mem, log_name='monomer1')

    # RESP charge calculation of a termination unit
    ter = utils.mol_from_smiles(ter_smiles)
    qm.assign_charges(ter, charge='RESP', opt=True, work_dir=work_dir, omp=omp_psi4, memory=mem, log_name='ter1')

    # Generate polymer chain
    dp = poly.calc_n_from_num_atoms(mol, 1000, terminal1=ter)
    homopoly = poly.polymerize_rw(mol, dp, tacticity='atactic')
    homopoly = poly.terminate_rw(homopoly, ter)
```

Sample code (continued)

```
# Force field assignment
result = ff.ff_assign(homopoly)
if not result:
    print('[ERROR:  Can not assign force field parameters.]')

# Generate simulation cell
ac = poly.amorphous_cell(homopoly,  10, density=0.05)

# Equilibration MD
eqmd = eq.EQ21step(ac,  work_dir=work_dir)
ac = eqmd.exec(temp=temp, press=1.0, mpi=mpi, omp=omp, gpu=gpu)

analy = eqmd.analyze()
prop_data = analy.get_all_prop(temp=temp,  press=1.0, save=True)
result = analy.check_eq()

# Additional equilibration  MD
for i in range(4):
    if result: break
    eqmd  = eq.Additional(ac,  work_dir=work_dir)
    ac = eqmd.exec(temp=temp,  press=press, mpi=mpi, omp=omp,  gpu=gpu)
    analy  = eqmd.analyze()
    prop_data  = analy.get_all_prop(temp=temp,  press=press, save=True)
    result = analy.check_eq()

if not result:
    print('[ERROR:  Did not reach an equilibrium  state.]')

# Non-equilibrium  MD for thermal  condultivity
else:
    nemd  = tc.NEMD_MP(ac, work_dir=work_dir)
    ac = nemd.exec(decomp=True,  temp=temp, mpi=mpi, omp=omp,  gpu=gpu)

    nemd_analy  = nemd.analyze()
    TC  = nemd_analy.calc_tc(decomp=True,  save=True)
    if not nemd_analy.Tgrad_data['Tgrad_check']:
        print('[ERROR:  Low linearity of temperature  gradient.]')

    print('Thermal  conductivity: %f' % TC)
```

Calculation results of physical properties

Appendix

Calculations of physical properties

Specific heat capacity

$$Cp = \frac{\langle \delta H^2 \rangle}{m k_B T^2}$$

Static dielectric constant

$$\varepsilon = 1 + \frac{\langle \boldsymbol{\mu}^2 \rangle - \langle \boldsymbol{\mu} \rangle^2}{3 \varepsilon_0 V k_B T^2}$$

Compressibility

$$\beta_T = \frac{\langle \delta V^2 \rangle}{V k_B T}$$

Bulk modulus

$$K_T = \frac{1}{\beta_T}$$

Thermal expansion coefficient

$$\alpha_P = \frac{\langle \delta H \cdot \delta V \rangle}{V k_B T^2}$$

Linear expansion coefficient

$$\alpha_{lx} = \frac{\alpha_P}{3} \quad \text{(isotropic)}$$

$$\alpha_{lx} = \frac{\langle \delta H \cdot \delta L_x \rangle}{L_x k_B T^2} \quad \text{(anisotropic)}$$

Self-diffusion coefficient

$$D = \frac{\langle |\boldsymbol{x}(t + t_0) - \boldsymbol{x}(t_0)|^2 \rangle}{6t}$$

Refractive index

$$\frac{n^2 - 1}{n^2 + 2} = \frac{4\pi}{3} \frac{\rho}{M} \alpha_{polar}$$

Unit of physical property in RadonPy

Physical property	Unit
Density	g/cm^3
Radius of gyration	Å
Cp, Cv	J/kg K
Compressibility	Pa^{-1}
Bulk modulus	Pa
Volume expansion coeff., Linear expansion coeff.	K^{-1}
Self-diffusion coeff.	m^2/s
Thermal conductivity	W/m K
Thermal diffusivity	m^2/s