Interfacing pyUNIxMD with external packages

Daeho Han

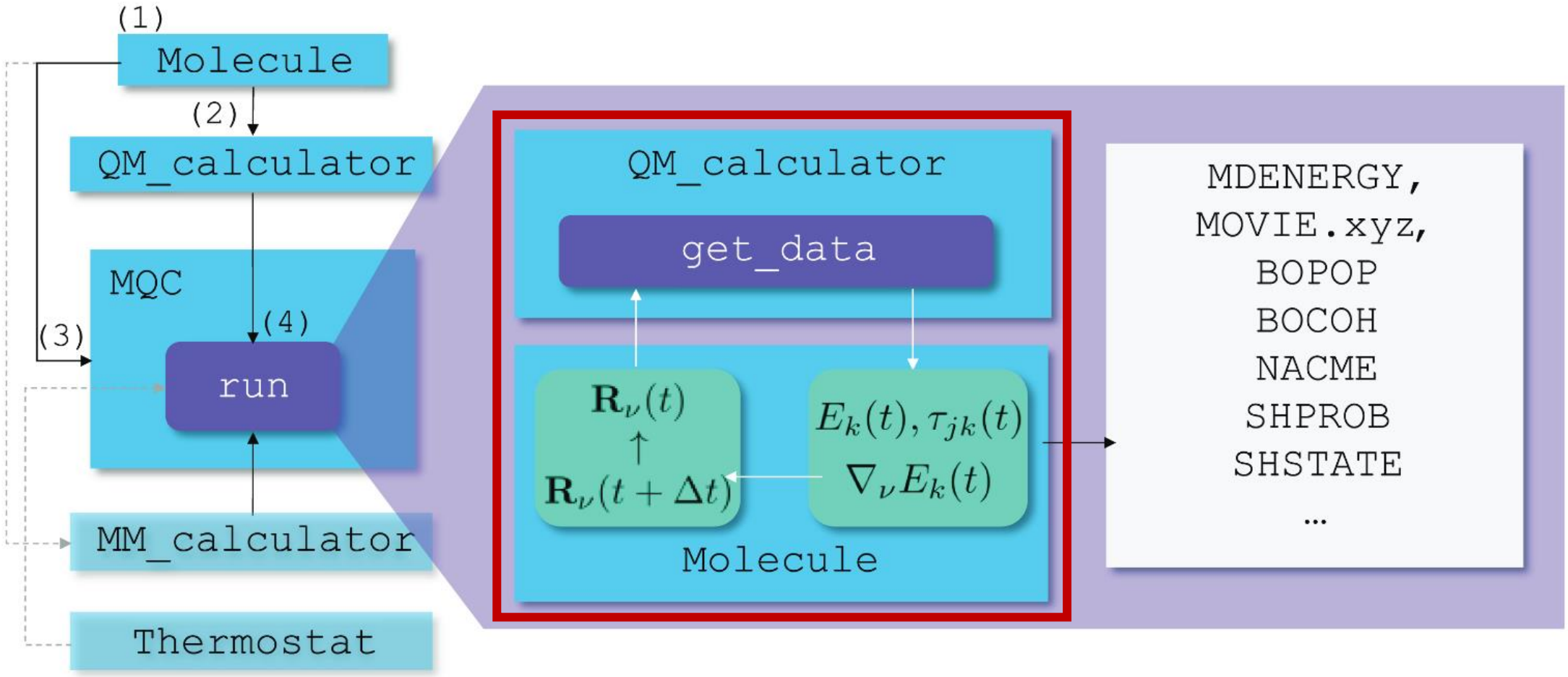
July 11, 2022

CONTACT

Excited State Phenomena Computational Chemistry Lab.
Office/Lab. 108) 805 **Tel.** +82 52 217 2918
Web. http://skmin.unist.ac.kr/

QM objects in pyUNIxMD

- How QM objects in pyUNIxMD handle with **BO energies, their gradients, NACs**?

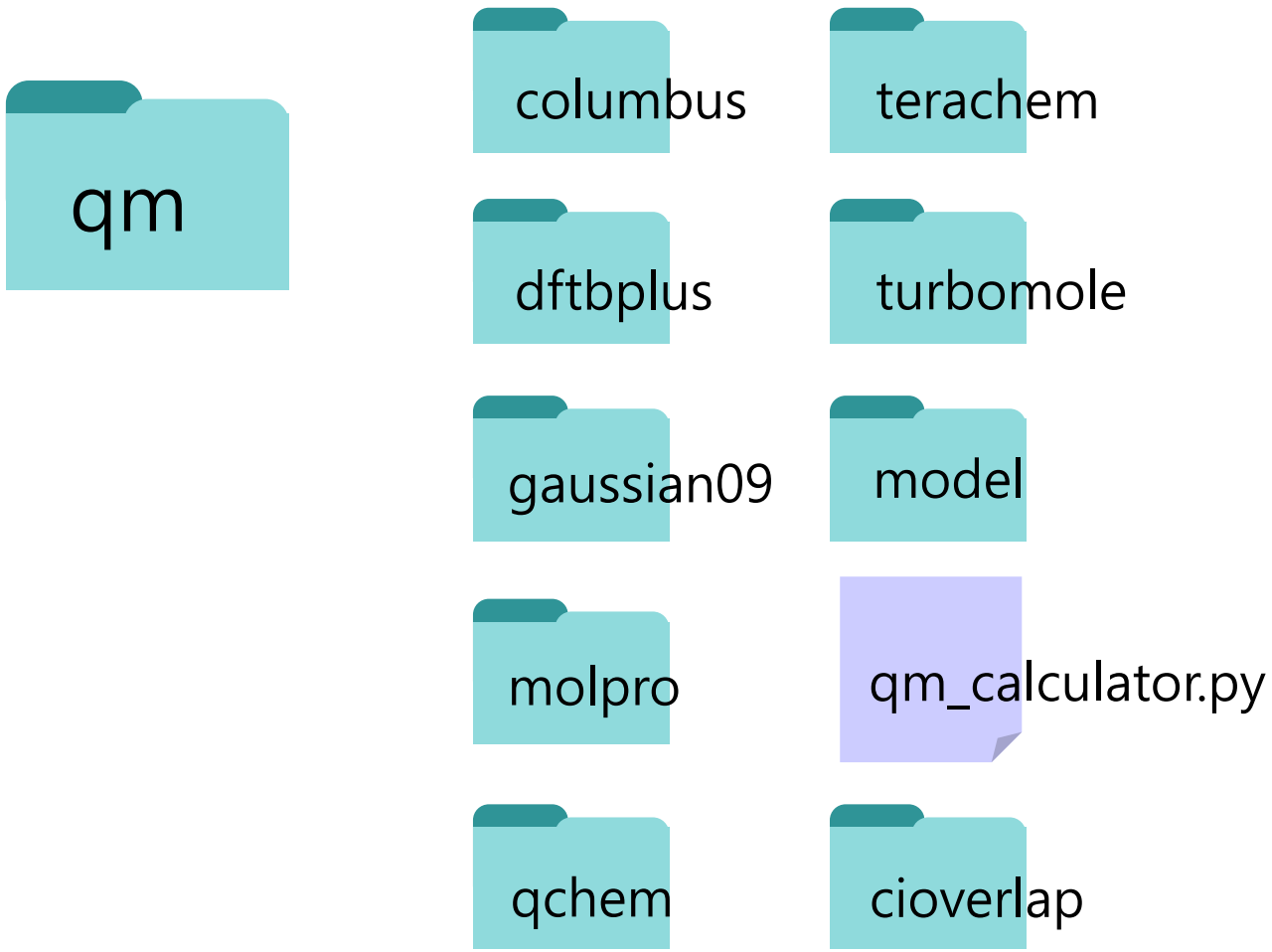


Typical workflow in pyUNIxMD

QM objects in pyUNIxMD

- QM packages in pyUNIxMD

/your-path-of-pyunixmd/src/qm

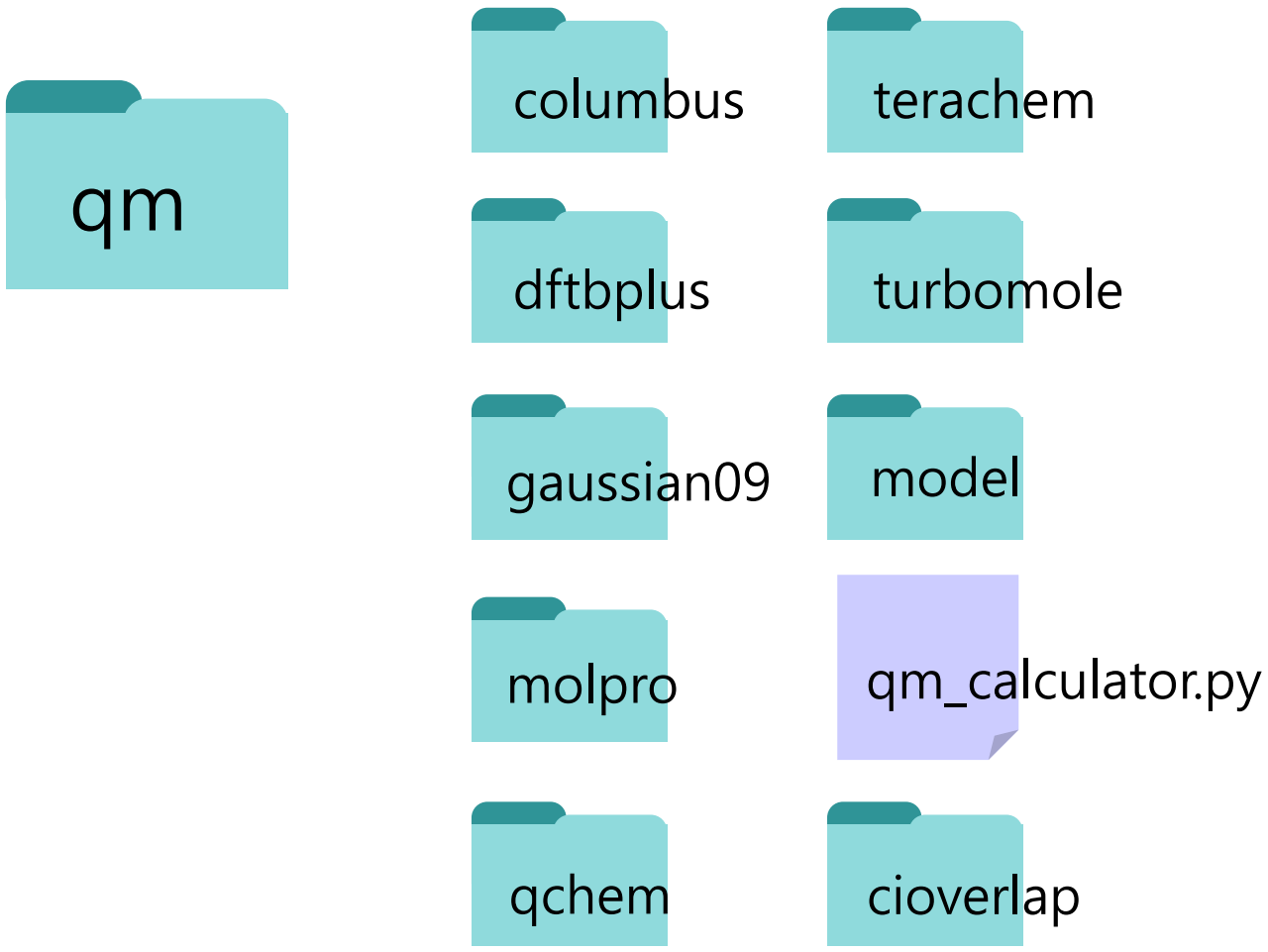


- For each external QM software there is a corresponding directory.
- qm_calculator.py defines a base class for QM objects.

QM objects in pyUNIxMD

- QM packages in pyUNIxMD

/your-path-of-pyunixmd/src/qm



qm_calculator.py

```
from __future__ import division
from misc import au_to_A
import os, shutil

class QM_calculator(object):
    """ Class for quantum mechanics calculator such as QM, ML, etc
    """
    def __init__(self):
        # Save name of QM calculator and its method
        self.qm_prog = str(self.__class__).split('.')[1]
        self.qm_method = self.__class__.__name__

    def get_data(self, base_dir, calc_force_only):
        """ Make scratch directory and copy geometry file

            :param string base_dir: Base directory
            :param boolean calc_force_only: Logical to decide whether
calculate force only
        """
        # Make 'scr_qm' directory
        unixmd_dir = os.path.join(base_dir, "md")
        self.scr_qm_dir = os.path.join(unixmd_dir, "scr_qm")
        if (not calc_force_only):
            if (os.path.exists(self.scr_qm_dir)):
                shutil.rmtree(self.scr_qm_dir)
            os.makedirs(self.scr_qm_dir)
        # Move to the scratch directory
        os.chdir(self.scr_qm_dir)
                                        ⋮
```

QM objects in pyUNIxMD

- QM packages in pyUNIxMD

/your-path-of-pyunixmd/src/qm/dftbplus

dftbplus.py –*DFTBplus* class inherited by *QM_calculator*

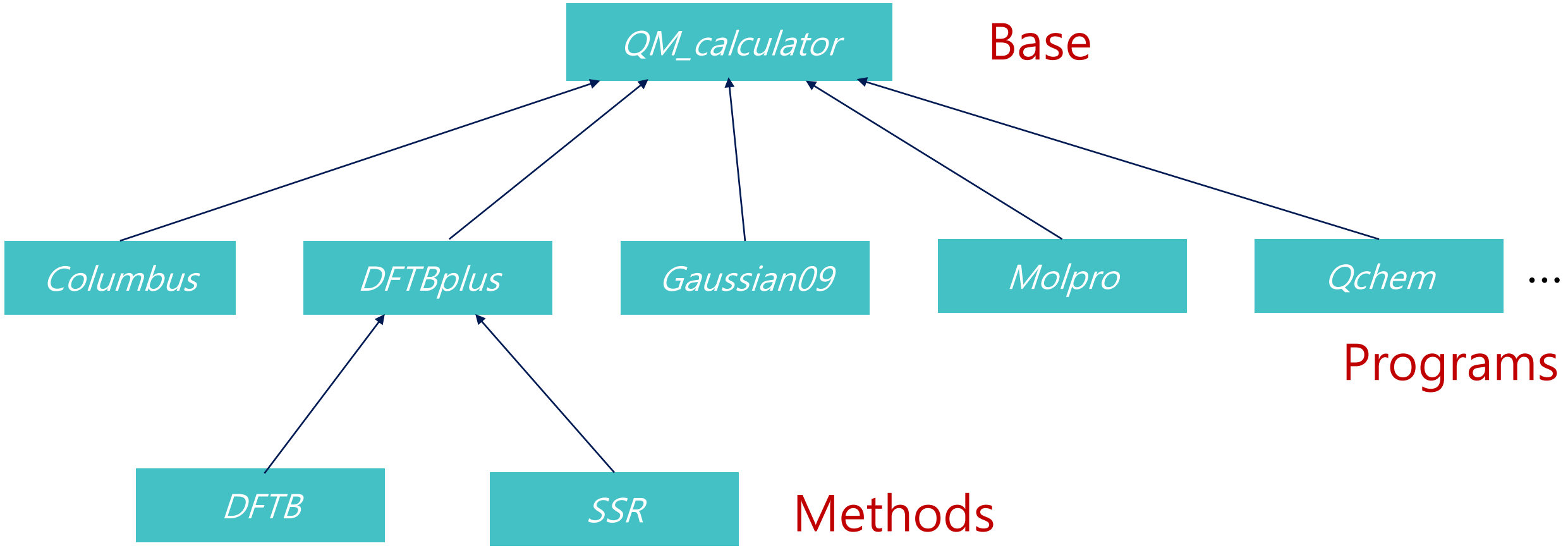
dftb.py – Class for TD-DFTB method inherited by *DFTBplus*

ssr.py – Class for DFTB/SSR method inherited by *DFTBplus*

dftbpar.py – Some parameters used in DFTB calculations

QM objects in pyUNIxMD

- QM packages in pyUNIxMD



Base-Programs-Methods inheritance relations

QM objects in pyUNIxMD

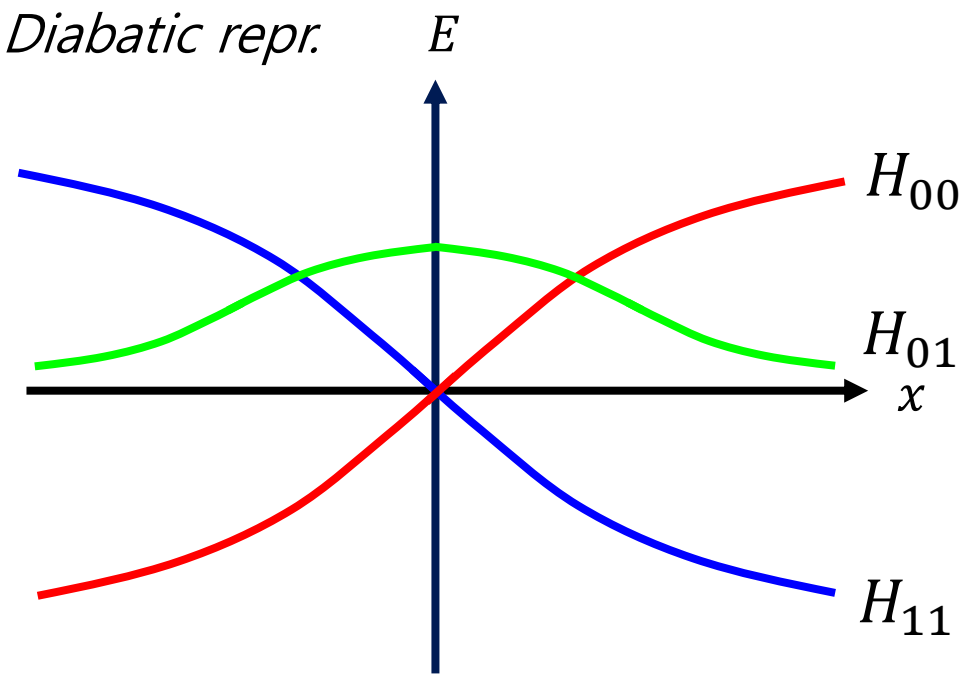
- DIY: Add a model to pyUNIxMD

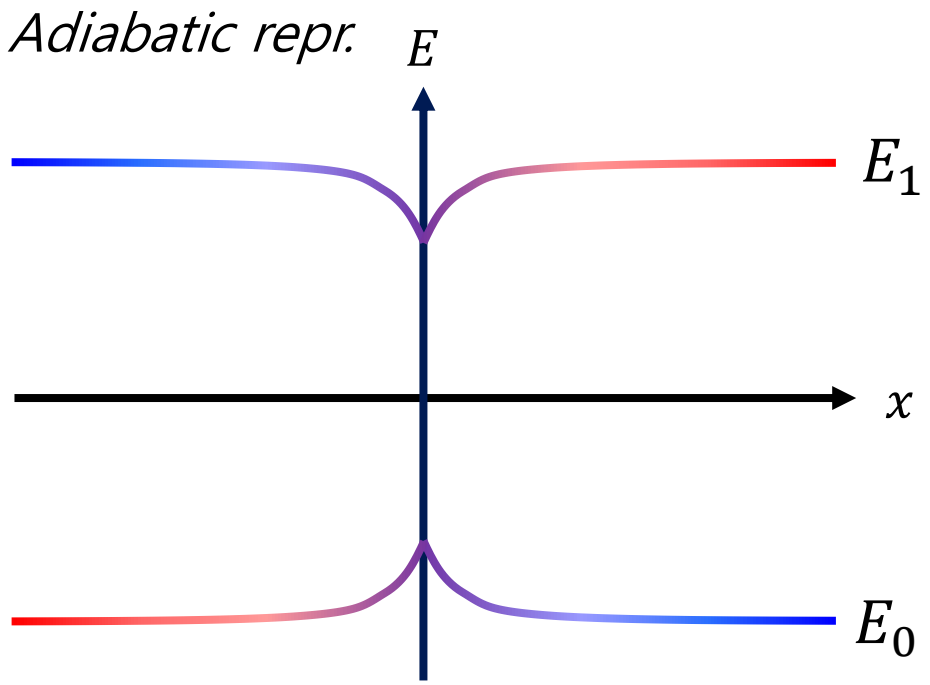
Tutorials_pyunixmd/add_a_new_model/

1. Fill in tully.py
2. Copy tully.py to /your-path-of-pyunixmd/src/qm/model/
3. Edit __init__.py in the above directory to make tully.py as one of the QM package.
4. Run `pes.py > pes.dat` to check the answer

/your-path-of-pyunixmd/src/qm/model/__init__.py

```
from .shin_metiu import Shin_Metiu
from .sac import SAC
from .dac import DAC
from .ecr import ECR
from .dag import DAG
from .tully import Tully
```
← Add this line!





QM objects in pyUNIxMD

- DIY: Add a model to pyUNIxMD

```
from __future__ import division
from qm.model.model import Model
import numpy as np

class Tully(Model):
    """ Class for simple avoided crossing (SAC) model BO calculation

        :param object molecule: molecule object
        :param double A: parameter for simple avoided crossing model
        :param double B: parameter for simple avoided crossing model
        :param double C: parameter for simple avoided crossing model
        :param double D: parameter for simple avoided crossing model
    """
    def __init__(self, molecule, A=0.01, B=1.6, C=0.005, D=1.):
        # Initialize model common variables
        super(Tully, self).__init__(None)

        # Define parameters
        self.A = A
        self.B = B
        self.C = C
        self.D = D

        # Set 'l_nacme' with respect to the computational method
        # SAC model can produce NACs, so we do not need to get NACME
        molecule.l_nacme = False

        # SAC model can compute the gradient of several states simultaneously
        self.re_calc = False
```

⋮

Tutorials_pyunixmd/add_a_new_model/

$$H_{00}(x) = A[1 - \exp(-Bx)], \quad x > 0$$

$$H_{00}(x) = -A[1 - \exp(Bx)], \quad x < 0$$

$$H_{11}(x) = -H_{00}(x)$$

$$H_{01}(x) = H_{10}(x) = C\exp(-Dx^2)$$

QM objects in pyUNIxMD

- DIY: Add a model to pyUNIxMD

Tutorials_pyunixmd/add_a_new_model/

```
def get_data(self, molecule, base_dir, bo_list, dt, istep, calc_force_only):
        """ Extract energy, gradient and nonadiabatic couplings from simple avoided crossing
model BO calculation

            :param object molecule: molecule object
            :param string base_dir: base directory
            :param integer,list bo_list: list of BO states for BO calculation
            :param double dt: time interval
            :param integer istep: current MD step
            :param boolean calc_force_only: logical to decide whether calculate force only
        """
        # Initialize diabatic Hamiltonian
        H = np.zeros((2, 2))
        dH = np.zeros((2, 2))
        U = np.zeros((2, 2))

        x = molecule.pos[0]

        # Define Hamiltonian

        # Define a derivative of Hamiltonian

        # Diagonalization

        # Extract adiabatic quantities
        molecule.states[0].energy =
        molecule.states[1].energy =

        molecule.states[0].force =
        molecule.states[1].force =

        molecule.nac[0, 1, 0, 0] =
        molecule.nac[1, 0, 0, 0] =
```

$$H_{00}(x) = A[1 - \exp(-Bx)], \quad x > 0$$

$$H_{00}(x) = -A[1 - \exp(Bx)], \quad x < 0$$

$$H_{11}(x) = -H_{00}(x)$$

$$H_{01}(x) = H_{10}(x) = C\exp(-Dx^2)$$

1. Use a formula for a 2X2 Hermitian matrix eigenvalue problem.

2. Use a Numpy eigensolver, np.linalg.eig.

$$U^{\dagger}HU = E \quad HU = H[\mathbf{u_0}\ \mathbf{u_1}] = UE$$

$$E = \begin{bmatrix} E_0 & 0 \\ 0 & E_1 \end{bmatrix} \quad F_0 = \mathbf{u_0}^{\dagger}\frac{dH}{dx}\mathbf{u_0} \quad F_1 = \mathbf{u_1}^{\dagger}\frac{dH}{dx}\mathbf{u_1}$$

$$\tau_{01} = \frac{\mathbf{u_0}^{\dagger}\frac{dH}{dx}\mathbf{u_1}}{E_1 - E_0} = -\tau_{10}$$

QM objects in pyUNIxMD

- DIY: Add a model to pyUNIxMD

pes.py Tutorials_pyunixmd/add_a_new_model/

```
from molecule import Molecule
import qm
from misc import data, amu_to_au

data["XX"] = 2000/amu_to_au

# Define the target system.
geom = """
1

XX      0.000000      0.000000
"""
mol = Molecule(geometry=geom, nstates=2, ndim=1, l_model=True)

for i, x in enumerate([x for x in range(100)]):
    mol.pos[0, 0] = -10 + 0.2*x

    # Set QM method.
    qm1 = qm.model.Tully(molecule=mol)

    qm1.get_data(mol, "", [], 0., -1, False)
    if (i > 0): mol.adjust_nac()

    mol.backup_bo()

    print(f"{mol.pos[0, 0]:13.8f} {mol.states[0].energy:13.8f}
{mol.states[1].energy:13.8f}\
 {mol.states[0].force:13.8f} {mol.states[1].force:13.8f} {mol.nac[0, 1, 0, 0]:13.8f}")
```

QM objects in pyUNIxMD

- DIY: Add a model to pyUNIxMD

Tutorials_pyunixmd/add_a_new_model/

1. Fill in tully.py
2. Copy tully.py to /your-path-of-pyunixmd/src/qm/model/
3. Edit __init__.py in the above directory to make tully.py as one of the QM package.
4. Run `pes.py > pes.dat` to check the answer

/your-path-of-pyunixmd/src/qm/model/__init__.py

```
from .shin_metiu import Shin_Metiu
from .sac import SAC
from .dac import DAC
from .ecr import ECR
from .dag import DAG
from .tully import Tully ← Add this line!
```

$$H_{00}(x) = A[1 - \exp(-Bx)], \quad x > 0$$

$$H_{00}(x) = -A[1 - \exp(Bx)], \quad x < 0$$

$$H_{11}(x) = -H_{00}(x)$$

$$H_{01}(x) = H_{10}(x) = C\exp(-Dx^2)$$

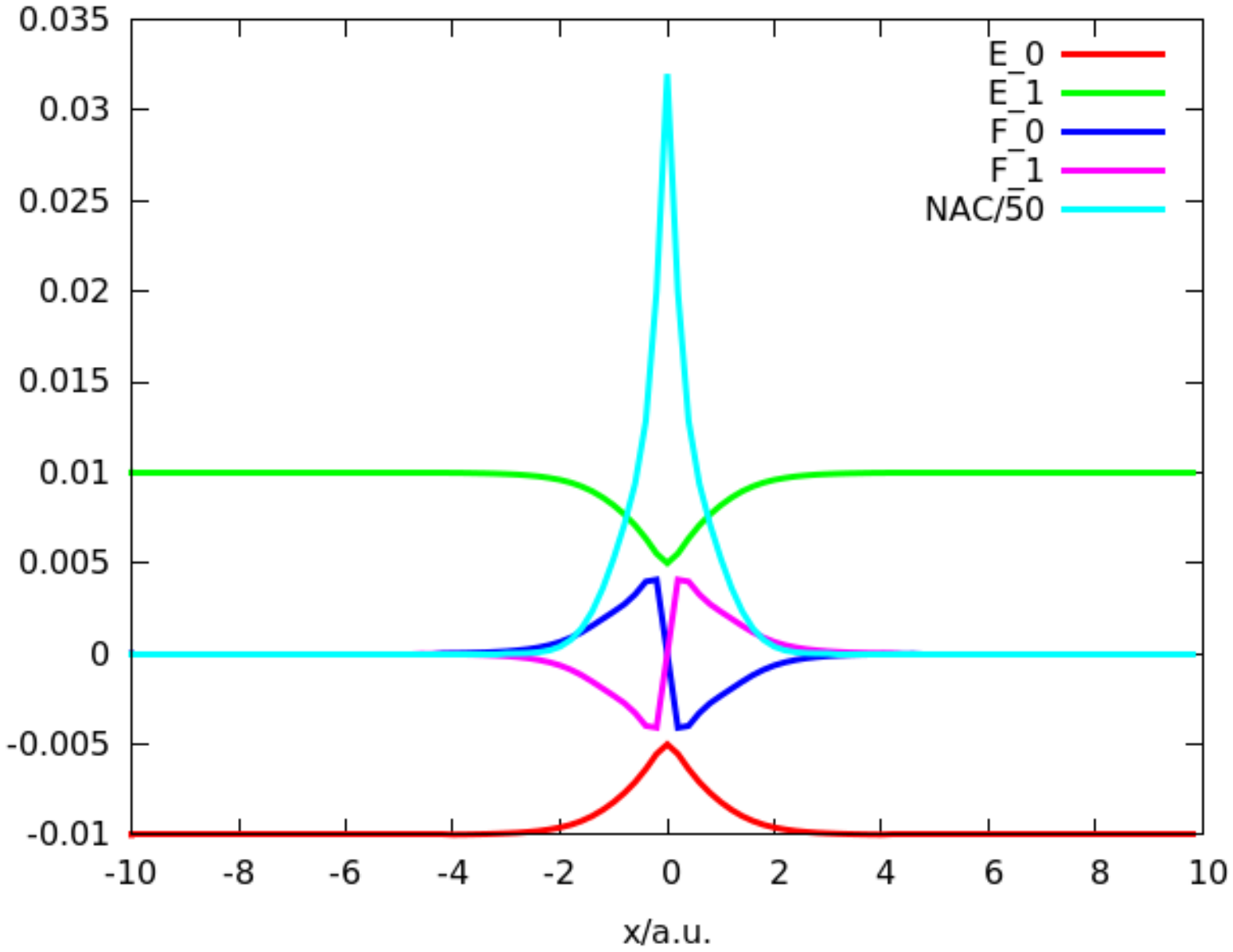
$$U^\dagger H U = E \qquad HU = H[\mathbf{u_0}\ \mathbf{u_1}] = UE$$

$$E = \begin{bmatrix} E_0 & 0 \\ 0 & E_1 \end{bmatrix} \quad F_0 = \mathbf{u_0^\dagger}\frac{dH}{dx}\mathbf{u_0} \quad F_1 = \mathbf{u_1^\dagger}\frac{dH}{dx}\mathbf{u_1}$$

$$\tau_{01} = \frac{\mathbf{u_0^\dagger}\frac{dH}{dx}\mathbf{u_1}}{E_1 - E_0} = -\tau_{10}$$

QM objects in pyUNIxMD

- DIY: Add a model to pyUNIxMD



From draw_dat.gpl

QM objects in pyUNIxMD

- DIY: Possible answers

get_data method in tully.py

```
…
# Define Hamiltonian
H[0, 0] = np.sign(x) * self.A * (1. - np.exp(- self.B * abs(x)))
H[1, 1] = - H[0, 0]
H[1, 0] = self.C * np.exp(- self.D * x ** 2)
H[0, 1] = H[1, 0]

# Define a derivative of Hamiltonian
dH[0, 0] = self.A * self.B * np.exp(- self.B * abs(x))
dH[1, 1] = - dH[0, 0]
dH[1, 0] = - 2. * self.D * self.C * x * np.exp(- self.D * x ** 2)
dH[0, 1] = dH[1, 0]

# Diagonalization
E = np.sqrt(H[0, 0] ** 2 + H[0, 1] ** 2)
U[:, 0] = np.array([H[0, 1], - E - H[0, 0]]).T
U[:, 1] = np.array([H[0, 1], E - H[0, 0]]).T

U[:, 0] /= np.linalg.norm(U[:, 0])
U[:, 1] /= np.linalg.norm(U[:, 1])

# Extract adiabatic quantities
molecule.states[0].energy = - E
molecule.states[1].energy = E

molecule.states[0].force = np.dot(U[:, 0].T, np.dot(dH, U[:, 0]))
molecule.states[1].force = np.dot(U[:, 1].T, np.dot(dH, U[:, 1]))

molecule.nac[0, 1, 0, 0] = np.dot(U[:, 0].T, np.dot(dH, U[:, 1])) / (2.*E)
molecule.nac[1, 0, 0, 0] = -np.copy(molecule.nac[0, 1, 0, 0])
```

Let $H = \begin{bmatrix} a & b \\ b & -a \end{bmatrix}$

Solve the characteristic equation to obtain eigenvalues.

$$|H - EI| = 0$$

$$(a - E)(-a - E) - b^2 = 0 \quad -a^2 + E^2 - b^2 = 0$$

$$E_{\pm} = \pm\sqrt{a^2 + b^2}$$

Let an eigenvector to be

$$\mathbf{u}_E = [c_1 \quad c_2]^{\dagger}$$

$$ac_1 + bc_2 = Ec_1 \qquad (a - E)c_1 + bc_2 = 0$$

$$ac_1 - bc_2 = Ec_2 \qquad ac_1 - (b + E)c_2 = 0$$

Thus, the unnormalized solutions are to be

$$\tilde{\mathbf{u}}_E = [b \quad E - a]^{\dagger} \quad \text{or} \quad \tilde{\mathbf{u}}_E = [E + a \quad b]^{\dagger}$$

Pick either and normalize to make $\mathbf{u}_{E_{\pm}}$.

QM objects in pyUNIxMD

- DIY: Possible answers

get_data method in tully.py

```
…
# Define Hamiltonian
H[0, 0] = np.sign(x) * self.A * (1. - np.exp(- self.B * abs(x)))
H[1, 1] = - H[0, 0]
H[1, 0] = self.C * np.exp(- self.D * x ** 2)
H[0, 1] = H[1, 0]

# Define a derivative of Hamiltonian
dH[0, 0] = self.A * self.B * np.exp(- self.B * abs(x))
dH[1, 1] = - dH[0, 0]
dH[1, 0] = - 2. * self.D * self.C * x * np.exp(- self.D * x ** 2)
dH[0, 1] = dH[1, 0]

# Diagonalization
E, U = np.linalg.eig(H)
idx = np.argsort(E)
E = E[idx]
U = U[:, idx]

# Extract adiabatic quantities
molecule.states[0].energy = E[0]
molecule.states[1].energy = E[1]

molecule.states[0].force = np.dot(U[:, 0].conj().T, np.dot(dH, U[:, 0]))
molecule.states[1].force = np.dot(U[:, 1].conj().T, np.dot(dH, U[:, 1]))

molecule.nac[0, 1, 0, 0] = np.dot(U[:, 0].conj().T, np.dot(dH, U[:, 1])) / (E[1] - E[0])
molecule.nac[1, 0, 0, 0] = -np.copy(molecule.nac[0, 1, 0, 0])
```

The eigenvalues and the corresponding eigenvectors are not sorted automatically.

QM objects in pyUNIxMD

- External package examples: DFTB+

/your-path-of-pyunixmd/src/qm/dftbplus

DFTBplus class

A program class has parameters related to the path, the version and the number of threads for a parallel calculation.

```
from __future__ import division
from qm.qm_calculator import QM_calculator
from misc import call_name
import os

class DFTBplus(QM_calculator):
    """ Class for common parts of DFTB+

        :param object molecule: Molecule object
        :param string sk_path: Path for Slater-Koster files
        :param string install_path: Path for DFTB+ install directory
        :param integer nthreads: Number of threads in the calculations
        :param string version: Version of DFTB+
    """
    def __init__(self, molecule, sk_path, install_path, nthreads, version):
        # Save name of QM calculator and its method
        super().__init__()

        # Initialize DFTB+ common variables
        self.sk_path = sk_path
        self.install_path = install_path
        if (not os.path.isdir(self.install_path)):
            error_message = "Install directory for DFTB+ not found!"
            error_vars = f"install_path = {self.install_path}"
            raise FileNotFoundError (f"( {self.qm_method}.{call_name()} ) {error_message}
( {error_vars} )")

        self.nthreads = nthreads
        self.version = version
```

QM objects in pyUNIxMD

- External package examples: DFTB+

/your-path-of-pyunixmd/src/qm/dftbplus

DFTBplus class

```
# Environmental variable setting for Python scripts such as xyz2gen used in DFTB+
if (isinstance(self.version, str)):
    if (self.version in ["19.1", "20.1", "21.1"]):
        self.qm_path = os.path.join(self.install_path, "bin")
        # Note that the Python version can be changed according to the users setting
        lib_dir = os.path.join(self.install_path, "lib/python3.6/site-packages")
        if (not os.path.exists(lib_dir)):
            error_message = "Please use proper Python version number in
'$PYUNIXMDHOME/src/qm/dftbplus/dftbplus.py'!"
            error_vars = f"library directory = {lib_dir}"
            raise FileNotFoundError (f"( {self.qm_method}.{call_name()} ) {error_message}
( {error_vars} )")
    else:
        error_message = "Other versions not implemented!"
        error_vars = f"version = {self.version}"
        raise ValueError (f"( {self.qm_method}.{call_name()} ) {error_message} ( {error_vars} )")
else:
    error_message = "Type of version must be string!"
    error_vars = f"version = {self.version}"
    raise TypeError (f"( {self.qm_method}.{call_name()} ) {error_message} ( {error_vars} )")

# Append following paths to PATH and PYTHONPATH variables
os.environ["PATH"] += os.pathsep + os.path.join(self.qm_path)
os.environ["PYTHONPATH"] += os.pathsep + os.path.join(lib_dir)

# Check the atomic species
self.atom_type = set(molecule.symbols[0:molecule.nat_qm])
```

Necessary environment variables are set.

Ex. the installed DFTB+ package path and DFTB+ python library path

QM objects in pyUNIxMD

- External package examples: DFTB+

/your-path-of-pyunixmd/src/qm/dftbplus

Constructor of DFTB class

A QM method object is initialized with input content for the corresponding QM program.

```
from __future__ import division
from build.cioverlap import *
from qm.dftbplus.dftbplus import DFTBplus
from qm.dftbplus.dftbpar import spin_w, spin_w_lc, onsite_uu, onsite_ud, max_l
from misc import data, eps, eV_to_au, call_name
import os, shutil, re, textwrap
import numpy as np

class DFTB(DFTBplus):
    """ Class for (TD)DFTB method of DFTB+

        :param object molecule: Molecule object
        :param boolean l_scc: Include self-consistent charge (SCC) scheme
        :param double scc_tol: Stopping criteria for the SCC iterations
        :param integer scc_max_iter: Maximum number of SCC iterations
        :param boolean l_onsite: Include onsite correction to SCC term
…
…
 def __init__(self, molecule, l_scc=True, scc_tol=1E-6, scc_max_iter=100, l_onsite=False, \
        l_range_sep=False, lc_method="MatrixBased", l_spin_pol=False, unpaired_elec=0., guess="h0", \
        guess_file="./charges.bin", elec_temp=0., mixer="Broyden", ex_symmetry="singlet", e_window=0., \
        k_point=[1, 1, 1], l_periodic=False, cell_length=[0., 0., 0., 0., 0., 0., 0., 0., 0.,], \
        sk_path="./", install_path="./", mpi=False, mpi_path="./", nthreads=1, version="20.1"):
        # Initialize DFTB+ common variables
        super(DFTB, self).__init__(molecule, sk_path, install_path, nthreads, version)

        # Initialize DFTB+ DFTB variables
        self.l_scc = l_scc
        self.scc_tol = scc_tol
        self.scc_max_iter = scc_max_iter

        self.l_onsite = l_onsite
…
…
```

QM objects in pyUNIxMD

- External package examples: DFTB+

/your-path-of-pyunixmd/src/qm/dftbplus

Constructor of DFTB class

In constructor of the QM method class, molecule.l_nacme and self.re_calc must be defined.

```
        self.a_axis = np.copy(cell_length[0:3])
        self.b_axis = np.copy(cell_length[3:6])
        self.c_axis = np.copy(cell_length[6:9])

        # Check excitation symmetry in TDDFTB
        # TODO : Currently, allows only singlet excited states with TDDFTB
#         if not (self.ex_symmetry in ["singlet", "triplet"]):
        if (not self.ex_symmetry == "singlet"):
            error_message = "Invalid symmetry of excited states for TDDFTB given!"
            error_vars = f"ex_symmetry = {self.ex_symmetry}"
            raise ValueError (f"( {self.qm_method}.{call_name()} ) {error_message}
( {error_vars} )")

        self.mpi = mpi
        self.mpi_path = mpi_path

        # Set 'l_nacme' and 're_calc' with respect to the computational method
        # TDDFTB do not produce NACs, so we should get NACME from CIoverlap
        # TDDFTB cannot compute the gradient of several states simultaneously.
        molecule.l_nacme = True
        self.re_calc = True
```

Analytical NACs are given. -> molecule.l_nacme := False ⇔ Read NACs from the output of QM program.
not given. -> molecule.l_nacme := True ⇔ Read orbitals (eigenvectors) and calculate overlap to calculate NACME by finite difference.

Only adiabatic force of an active state is calculated. -> self.re_calc := True ⇔ You need to recalculate a force when a hop occurs.
All adiabatic forces are calculated. -> self.re_calc := False ⇔ You don't have to recalculate because you already have what you need.

QM objects in pyUNIxMD

- External package examples: DFTB+

/your-path-of-pyunixmd/src/qm/dftbplus

Constructor of DFTB class

When molecule.l_nacme = True, that is, NAC calculation with finite difference is done, related variables are needed to be initialized.

```
 …
# Initialize NACME variables
        # There is no core orbitals in TDDFTB (fixed occupations)
        # nocc is number of occupied orbitals and nvirt is number of virtual orbitals
        self.norb = self.nbasis
        self.nocc = int(int(molecule.nelec - core_elec) / 2)
        self.nvirt = self.norb - self.nocc

        # Replace norb by arrays containing the limits of the for loops.
        # For energy window calculations loops will not go from (0 to nocc/nvirt) or (0 to norb)
        # but from (nocc_min to nocc/0 to nvirt_max) or (nocc_min to norb).
        self.orb_ini = np.zeros(1, dtype=np.int32)
        self.orb_final = np.zeros(1, dtype=np.int32)
        self.orb_final[0] = self.norb

        if (self.e_window > eps):
            # Swap minimal/maximal values to replace them in reading of SPX.DAT by the
minimal/maximal values.
            self.orb_ini[0] = self.norb
            self.orb_final[0] = 0

        self.ao_overlap = np.zeros((self.nbasis, self.nbasis))
        self.mo_coef_old = np.zeros((self.norb, self.nbasis))
        self.mo_coef_new = np.zeros((self.norb, self.nbasis))
        self.ci_coef_old = np.zeros((molecule.nst, self.nocc, self.nvirt))
        self.ci_coef_new = np.zeros((molecule.nst, self.nocc, self.nvirt))
```

$$\tau_{nk} = \sum_{ia} C_{ia}^{n} \partial_t C_{ia}^{k} + \sum_{iab} C_{ia}^{n} C_{ib}^{k} \langle \phi_a | \partial_t \phi_b \rangle - \sum_{ija} P_{ij} C_{ia}^{n} C_{ja}^{k} \langle \phi_j | \partial_t \phi_i \rangle$$

J. Phys. Chem. Lett. **2015**, 6, 21, 4200–4203.

QM objects in pyUNIxMD

- External package examples: DFTB+

/your-path-of-pyunixmd/src/qm/dftbplus

The get_data method of an QM method object do make input, run a calculation and extract data from the output (get_input, run_QM, and extract_QM method).

The get_data method of DFTB class

```
def get_data(self, molecule, base_dir, bo_list, dt, istep, calc_force_only):
        """ Extract energy, gradient and nonadiabatic couplings from (TD)DFTB method

            :param object molecule: Molecule object
            :param string base_dir: Base directory
            :param integer,list bo_list: List of BO states for BO calculation
            :param double dt: Time interval
            :param integer istep: Current MD step
            :param boolean calc_force_only: Logical to decide whether calculate force only
        """
        self.copy_files(molecule, istep, calc_force_only)
        super().get_data(base_dir, calc_force_only)
        self.write_xyz(molecule)
        self.get_input(molecule, istep, bo_list, calc_force_only)
        self.run_QM(molecule, base_dir, istep, bo_list, calc_force_only)
        self.extract_QM(molecule, base_dir, istep, bo_list, dt, calc_force_only)
        self.move_dir(base_dir)
```

Excited State Phenomena Computational Chemistry Lab.