dqc Documentation

Release 0.2.0.dev92016679455

Muhammad Firmansyah Kasim

Mar 10, 2022

GETTING STARTED

DQC is a PyTorch-based quantum chemistry simulation software that can automatically provide gradients of (almost) any variables with respect to (almost) any variables. DQC provides analytic first and higher order derivatives automatically using PyTorch's autograd engine.

DQC's example API:

```
import torch
import dqc
atomzs, atomposs = dqc.parse_moldesc("H -1 0 0; H 1 0 0")
atomposs = atomposs.requires_grad_()  # mark atomposs as differentiable
mol = dqc.Mol((atomzs, atomposs), basis="3-21G")
qc = dqc.HF(mol).run()
ene = qc.energy()  # calculate the energy
force = -torch.autograd.grad(ene, atomposs)[0]  # calculate the force
print(force)
```

```
tensor([[ 0.1033, -0.0000, -0.0000],
        [-0.1033, -0.0000, -0.0000]], dtype=torch.float64)
```

CHAPTER

ONE

INSTALLATION

1.1 Requirements

- python >= 3.7 with pip

1.2 Installation

Installation can be done as simple as typing in the terminal the line below:

```
python -m pip install dqc
```

To install the nightly version, you can type:

```
git clone https://github.com/diffqc/dqc
cd dqc
python -m pip install -e .
```

CHAPTER

TWO

HOW TO CONTRIBUTE

There are a lot of ways to contribute to DQC:

- Report bugs or request features via DQC Github page
- Improving documentations
- Reporting and fixing bugs
- Writing a new feature / simulation

2.1 Improving documentations

If you found a typo or you think you had a better way to explain things in the documentation, you can directly make a pull request to our Github page.

If you would like to see or write a tutorial on a certain things, please raise an issue to the issue page.

2.2 Reporting and fixing bugs

If you find a bug, please report it to the Github page with a simple example on how to reproduce the bug. You can fix the bug you reported or the bugs reported by others.

2.3 Writing a new feature / simulation

If you want to implement a new feature or simulation, please raise an issue on the Github page before starting your work. It will be reviewed if the feature is suitable for DQC. If approved, you are welcome to make a pull request. We can also give pointers on how to implement the feature if you don't know where to start.

CHAPTER

THREE

TUTORIALS

3.1 Calculating force

Calculating force in DQC is straightforward as shown by the code below

```
import torch
import dqc
atomzs = torch.tensor([1, 1])
atomposs = torch.tensor([[1, 0, 0], [-1, 0, 0]], dtype=torch.double).requires_grad_()
mol = dqc.Mol(moldesc=(atomzs, atomposs), basis="3-21G")
qc = dqc.HF(mol).run()
ene = qc.energy()  # calculate the energy
force = -torch.autograd.grad(ene, atomposs)[0]  # calculate the force
print(force)
```

```
The 3-21G basis for atomz 1 does not exist, but we will download it
Downloaded to /home/docs/checkouts/readthedocs.org/user_builds/dqc/envs/latest/lib/
↪python3.8/site-packages/dqc/api/.database/3-21g/01.gaussian94
tensor([[-0.1033, -0.0000, -0.0000],
        [ 0.1033, -0.0000, -0.0000]], dtype=torch.float64)
```

Let's take a look at each parts. First, we set up the atomic numbers and positions with lines:

```
atomzs = torch.tensor([1, 1])
atomposs = torch.tensor([[1, 0, 0], [-1, 0, 0]], dtype=torch.double).requires_grad_()
```

The system has two atoms with atomic number 1 and 1 (i.e. hydrogen molecule) as set by `atomzs` variable. In the next line, `atomposs` describes the position of each atom, i.e. the first one is at `(1, 0, 0)` and the second one is at `(-1, 0, 0)`. Please note that `atomposs` is marked as differentiable by `.requires_grad_()` command. This is required as we want to differentiate the energy later with respect to the atomic positions.

Next, we construct the DQC system by:

```
mol = dqc.Mol((atomzs, atomposs), basis="3-21G")
```

The first argument of *Mol* is molecular description which can accept a tuple of `(atomzs, atomposs)` or a string description (explained later). The `basis` keyword choose the basis for each atom. In this case, it uses `"3-21G"` basis set.

Once the DQC system is constructed, then we can run the calculation by

```
qc = dqc.HF(mol).run()
ene = qc.energy()  # calculate the energy
```

The first line above runs the simulation until it reaches convergence. Then, the next line calculates the energy. The output energy is differentiable with respect to floating point tensors in the system that are set to be differentiable. Therefore, the force can be simply calculated by

```
force = -torch.autograd.grad(ene, mol.atompos)[0]  # calculate the force
```

How if we have molecule description in string, e.g. `"H -1 0 0; H 1 0 0"`? In this case, we need a help from *parse_moldesc()*,

```
import torch
import dqc
atomzs, atomposs = dqc.parse_moldesc("H -1 0 0; H 1 0 0")
atomposs = atomposs.requires_grad_()  # marking atomposs as differentiable
mol = dqc.Mol(moldesc=(atomzs, atomposs), basis="3-21G")
qc = dqc.HF(mol).run()
ene = qc.energy()  # calculate the energy
force = -torch.autograd.grad(ene, atomposs)[0]  # calculate the force
print(force)
```

```
tensor([[ 0.1033, -0.0000, -0.0000],
        [-0.1033, -0.0000, -0.0000]], dtype=torch.float64)
```

The only difference in this case is the lines

```
atomzs, atomposs = dqc.parse_moldesc("H -1 0 0; H 1 0 0")
atomposs = atomposs.requires_grad_()  # marking atomposs as differentiable
```

where *parse_moldesc()* parses the string and returns two tensors describing the atomic numbers and atomic positions. The rest are just the same as the previous case.

3.2 Calculating gradients of xc functional

One of DQC applications is to optimize xc functional to fit some properties. In this tutorial, we will see how to get the gradient of custom xc functionals for the optimization of custom xc functionals.

First, we need to define our custom xc functional.

```
import torch
import dqc
import dqc.xc
import dqc.utils
class MyLDAX(dqc.xc.CustomXC):
    def __init__(self, a, p):
        super().__init__()
        self.a = a
        self.p = p

    @property
    def family(self):
        # 1 for LDA, 2 for GGA, 4 for MGGA
        return 1

    def get_edensityxc(self, densinfo):
        # densinfo has up and down components
        if isinstance(densinfo, dqc.utils.SpinParam):
```

(continues on next page)

(continued from previous page)

```
            # spin-scaling of the exchange energy
            return 0.5 * (self.get_edensityxc(densinfo.u * 2) + self.get_
↪edensityxc(densinfo.d * 2))
        else:
            rho = densinfo.value.abs() + 1e-15  # safeguarding from nan
            return self.a * rho ** self.p

a = torch.nn.Parameter(torch.tensor(1.0, dtype=torch.double))
p = torch.nn.Parameter(torch.tensor(2.0, dtype=torch.double))
myxc = MyLDAX(a, p)
```

The base class `CustomXC` is required to define a new xc functional. `CustomXC` is a child class of `torch.nn.Module`, so the initial `super().__init__()` is required. In our custom xc functional, only `get_edensityxc` that needs to be written, which calculates the xc energy density per volume, as well as specifying the family of the functional.

The `densinfo` input of `get_edensityxc` can be either: *SpinParam* or *ValGrad*. *SpinParam* is DQC data structure to store variables for spin up and spin down. *ValGrad* is another DQC data structure to save the density information by having attributes: `value` for local value, `grad` for local gradients, `lapl` for the local laplacian, and `kin` for the local kinetic energy.

Once the custom xc functional is defined, we can use it for DFT calculation.

```
mol = dqc.Mol(moldesc="H -1 0 0; H 1 0 0", basis="3-21G")
qc = dqc.KS(mol, xc=myxc).run()
ene = qc.energy()
print(ene)
```

```
tensor(-0.4645, dtype=torch.float64, grad_fn=<AddBackward0>)
```

And to get the gradient with respect to the xc parameters, it is straightforward.

```
grad_a, grad_p = torch.autograd.grad(ene, (a, p))
print(grad_a, grad_p)
```

```
tensor(0.0711, dtype=torch.float64) tensor(-0.2108, dtype=torch.float64)
```

3.3 Alchemical perturbation

In this tutorial, we will show how to estimate properties of molecules using alchemical perturbation. Specifically, we will estimate the distance between two atoms in CO and BF molecules by only calculating the properties of N_2 and its alchemical perturbation.

Let's denote the atomic number of the atoms in the diatomic molecule as

$$Z_{\pm}(\lambda) = 7 \pm \lambda$$

parameterized by a variable λ. One of the atom takes the plus sign while another one takes the minus sign to keep the number of electrons constant.

The equilibrium distance between the atoms is defined as

$$s^*(\lambda) = \arg\min_s \mathcal{E}(s, \lambda)$$

where $\mathcal{E}$ is the total energy as a function of the atomic distance s and λ. What we will do is to estimate the equilibrium distance for $\lambda = 1$ (for CO) and $\lambda = 2$ (for BF) using Taylor expansion,

$$s^*(\lambda) \approx s^*(0) + \lambda \frac{\partial s^*}{\partial \lambda} + \frac{1}{2}\lambda^2 \frac{\partial^2 s^*}{\partial \lambda^2}$$

As a demonstration, we will use Hartree-Fock calculation with 3-21G basis set. First, we need to import modules and set up variables that we will need for the calculations.

```
import torch
import dqc
import xitorch.optimize  # for differentiable optimization
dtype = torch.double
basis = dqc.loadbasis("7:3-21G")
```

```
The 3-21G basis for atomz 7 does not exist, but we will download it
Downloaded to /home/docs/checkouts/readthedocs.org/user_builds/dqc/envs/latest/lib/
↪python3.8/site-packages/dqc/api/.database/3-21g/07.gaussian94
```

`xitorch` is a great library that provides differentiable functionals that we will use in this tutorial. The last line with *`dqc.loadbasis()`* loads the basis 3-21G for atomic number 7. We will use the same basis for all values of λ to make sure there is no discontinuity in the properties.

Next, we need to define a function that calculates the energy given the distance s and λ.

```
def get_energy(s, lmbda):
    atomzs = 7.0 + torch.tensor([1.0, -1.0], dtype=dtype) * lmbda
    atomposs = torch.tensor([[-0.5, 0, 0], [0.5, 0, 0]], dtype=dtype) * s
    mol = dqc.Mol((atomzs, atomposs), spin=0, basis=[basis, basis])
    qc = dqc.HF(mol).run()
    return qc.energy()
```

Once the function is defined, then we can calculate the equilibrium distance for N_2 molecule.

```
lmbda = torch.tensor(0.0, dtype=dtype).requires_grad_()
s0_n2 = torch.tensor(2.04, dtype=dtype)  # initial guess of the distance
smin_n2 = xitorch.optimize.minimize(
    get_energy, s0_n2, params=(lmbda,), method="gd", step=1e-2)
print(smin_n2)
```

```
tensor(2.0460, dtype=torch.float64, grad_fn=<_RootFinderBackward>)
```

`xitorch.optimize.minimize` finds the parameters `s` that minimizes the energy given the parameters `lmbda`. The output of `xitorch.optimize.minimize` is now differentiable with respect to the parameter `lmbda`.

```
grad_lmbda = torch.autograd.grad(smin_n2, lmbda, create_graph=True)[0]
grad2_lmbda = torch.autograd.grad(grad_lmbda, lmbda, create_graph=True)[0]
print(grad_lmbda.detach(), grad2_lmbda.detach())
```

```
tensor(1.3323e-15, dtype=torch.float64) tensor(0.1323, dtype=torch.float64)
```

Now, we can estimate the equilibrium distance of CO and BF,

```
smin_co = smin_n2 + grad_lmbda + 0.5 * grad2_lmbda
smin_bf = smin_n2 + grad_lmbda * 2 + 0.5 * grad2_lmbda * 2 ** 2
print(smin_co.detach(), smin_bf.detach())
```

```
tensor(2.1121, dtype=torch.float64) tensor(2.3105, dtype=torch.float64)
```

For reference, the equilibrium distances for CO and BF by minimizing the energy are 2.1119 and 2.3103 Bohr, respectively, which are quite close to the values above.

CHAPTER

FOUR

SYSTEMS (DQC)

4.1 Mol

class dqc.**Mol**(*moldesc: Union[str, Tuple[Union[List[str], List[Union[int, float, torch.Tensor]], torch.Tensor], Union[List[List[float]], numpy.ndarray, torch.Tensor]]]*, *basis: Union[str, List[dqc.utils.datastruct.CGTOBasis], List[str], List[List[dqc.utils.datastruct.CGTOBasis]], Dict[Union[str, int], Union[List[dqc.utils.datastruct.CGTOBasis], str]]]*, ***, *orthogonalize_basis: bool = True*, *ao_parameterizer: str = 'qr'*, *grid: Union[int, str] = 'sg3'*, *spin: Optional[Union[int, float, torch.Tensor]] = None*, *charge: Union[int, float, torch.Tensor] = 0*, *orb_weights: Optional[dqc.utils.datastruct.SpinParam[torch.Tensor]] = None*, *efield: Optional[Union[torch.Tensor, Tuple[torch.Tensor, ...]]] = None*, *vext: Optional[torch.Tensor] = None*, *dtype: torch.dtype = torch.float64*, *device: torch.device = device(type='cpu')*)

Describe the system of an isolated molecule.

Parameters

- **moldesc** (*) – Description of the molecule system. If string, it can be described like `"H 1 0 0; H -1 0 0"`. If tuple, the first element of the tuple is the Z number of the atoms while the second element is the position of the atoms: `(atomzs, atomposs)`.
- **basis** (*) – The string describing the gto basis. If it is a list, then it must have the same length as the number of atoms.
- **grid** (*) – Describe the grid. If it is an integer, then it uses the default grid with specified level of accuracy.
- **spin** (*) – The difference between spin-up and spin-down electrons. It must be an integer or `None`. If `None`, then it is `num_electrons % 2`. For floating point atomzs and/or charge, the `spin` must be specified.
- **charge** (*) – The charge of the molecule.
- **orb_weights** (*) – Specifiying the orbital occupancy (or weights) directly. If specified, `spin` and `charge` arguments are ignored.
- **vext** (*) – The tensor describing the external potential given in the grid. The grid position can be obtained by `Mol().get_grid().get_rgrid()`.
- **efield** (*) – Uniform electric field of the system. If a tensor, then it is assumed to be a constant electric field with the energy is calculated based on potential at `(0, 0, 0)` is `0`. If a tuple of tensor, then the first element will have a shape of `(ndim,)` representing the constant electric field, second element is the gradient of electric field with the last dimension is the direction of the electric field, third element is the gradgrad of electric field, etc. If `None`, then the electric field is assumed to be `0`.

- **dtype** (*) – The data type of tensors in this class.
- **device** (*) – The device on which the tensors in this class are stored.
- **orthogonalize_basis** (*) – (computational option) If True, orthogonalize the basis in the hamiltonian calculation. If False, then use the raw basis, this might not work with over-complete basis.
- **ao_parameterizer** (*) – (computational option) Specifying the atomic orbital parameterizer.

densityfit(*method: Optional[str] = None, auxbasis: Optional[Union[str, List[dqc.utils.datastruct.CGTOBasis], List[str], List[List[dqc.utils.datastruct.CGTOBasis]], Dict[Union[str, int], Union[List[dqc.utils.datastruct.CGTOBasis], str]]]] = None*) → dqc.system.base_system.BaseSystem

Indicate that the system's Hamiltonian uses density fit for its integral.

Parameters

- **method** (*Optional[str]*) – Density fitting method. Available methods in this class are:
  - `"coulomb"`: Minimizing the Coulomb inner product, i.e. `min <p-p_fit|r_12|p-p_fit>` Ref: Eichkorn, et al. Chem. Phys. Lett. 240 (1995) 283-290. (default)
  - `"overlap"`: Minimizing the overlap inner product, i.e. min <p-p_fit|p-p_fit>
- **auxbasis** (*Optional[BasisInpType]*) – Auxiliary basis for the density fit. If not specified, then it uses `"cc-pvtz-jkfit"`.

get_hamiltonian() → dqc.hamilton.base_hamilton.BaseHamilton

Returns the Hamiltonian that corresponds to the system, i.e. *HamiltonCGTO*

set_cache(*fname: str, paramnames: Optional[List[str]] = None*) → dqc.system.base_system.BaseSystem

Setup the cache of some parameters specified by *paramnames* to be read/written on a file. If the file exists, then the parameters will not be recomputed, but just loaded from the cache instead. Cache is usually used for repeated calculations where the cached parameters are not changed (e.g. running multiple systems with slightly different environment.)

Parameters

- **fname** (*str*) – The file to store the cache.
- **paramnames** (*list of str or None*) – List of parameter names to be read/write from the cache.

get_orbweight(*polarized: bool = False*) → Union[torch.Tensor, dqc.utils.datastruct.SpinParam[torch.Tensor]]

Returns the atomic orbital weights. If polarized == False, then it returns the total orbital weights. Otherwise, it returns a tuple of orbital weights for spin-up and spin-down.

get_nuclei_energy() → torch.Tensor

Returns the nuclei-nuclei repulsion energy.

setup_grid() → None

Construct the integration grid for the system

get_grid() → dqc.grid.base_grid.BaseGrid

Returns the grid of the system

requires_grid() → bool
True if the system needs the grid to be constructed. Otherwise, returns False

getparamnames(*methodname: str*, *prefix: str = ''*) → List[str]
This method should list tensor names that affect the output of the method with name indicated in `methodname`. If the `methodname` is not on the list in this function, it should raise `KeyError`.

Parameters

- **methodname** (*str*) – The name of the method of the class.
- **prefix** (*str*) – The prefix to be appended in front of the parameters name. This usually contains the dots.

Returns Sequence of name of parameters affecting the output of the method.

Return type Sequence of string

Raises **KeyError** – If the list in this function does not contain `methodname`.

Example

```
>>> class A(xitorch.EditableModule):
...     def __init__(self, a):
...         self.b = a*a
...
...     def mult(self, x):
...         return self.b * x
...
...     def getparamnames(self, methodname, prefix=""):
...         if methodname == "mult":
...             return [prefix+"b"]
...         else:
...             raise KeyError()
```

make_copy(***kwargs*) → dqc.system.mol.Mol
Returns a copy of the system identical to the orginal except for new parameters set in the kwargs.

Parameters ****kwargs** – Must be the same kwargs as Mol.

property atompos
Returns the atom positions as a tensor with shape `(natoms, ndim)`

property atomzs
Returns the tensor containing the atomic number with shape `(natoms,)`

property atommasses
Returns the tensor containing atomic mass with shape `(natoms)` in atomic unit

property spin
Returns the total spin of the system.

property charge
Returns the charge of the system.

property numel
Returns the total number of the electrons in the system.

property efield
Returns the external electric field of the system, or None if there is no electric field.

4.2 Sol

class dqc.**Sol**(*soldesc: Union[str, Tuple[Union[List[str], List[Union[int, float, torch.Tensor]], torch.Tensor], Union[List[List[float]], numpy.ndarray, torch.Tensor]]]*, *alattice: torch.Tensor*, *basis: Union[str, List[dqc.utils.datastruct.CGTOBasis], List[str], List[List[dqc.utils.datastruct.CGTOBasis]]]*, ***, *grid: Union[int, str] = 'sg3'*, *spin: Optional[Union[int, float, torch.Tensor]] = None*, *lattsum_opt: Optional[Union[dqc.hamilton.intor.pbcintor.PBCIntOption, Dict]] = None*, *dtype: torch.dtype = torch.float64*, *device: torch.device = device(type='cpu')*)

Describe the system of a solid (i.e. periodic boundary condition system).

Parameters

- **soldesc** (***) – Description of the molecule system. If string, it can be described like `"H 1 0 0; H -1 0 0"`. If tuple, the first element of the tuple is the Z number of the atoms while the second element is the position of the atoms: `(atomzs, atomposs)`.
- **basis** (***) – The string describing the gto basis. If it is a list, then it must have the same length as the number of atoms.
- **grid** (***) – Describe the grid. If it is an integer, then it uses the default grid with specified level of accuracy.
- **spin** (***) – The difference between spin-up and spin-down electrons. It must be an integer or `None`. If `None`, then it is `num_electrons % 2`. For floating point atomzs and/or charge, the `spin` must be specified.
- **charge** (***) – The charge of the molecule.
- **orb_weights** (***) – Specifiying the orbital occupancy (or weights) directly. If specified, `spin` and `charge` arguments are ignored.
- **dtype** (***) – The data type of tensors in this class.
- **device** (***) – The device on which the tensors in this class are stored.

densityfit(*method: Optional[str] = None*, *auxbasis: Optional[Union[str, List[dqc.utils.datastruct.CGTOBasis], List[str], List[List[dqc.utils.datastruct.CGTOBasis]], Dict[Union[str, int], Union[List[dqc.utils.datastruct.CGTOBasis], str]]]] = None*) → dqc.system.base_system.BaseSystem

Indicate that the system's Hamiltonian uses density fit for its integral.

Parameters

- **method** (*Optional[str]*) – Density fitting method. Available methods in this class are:
  - **"gdf": Density fit with gdf compensating charge to perform** the lattice sum. Ref https://doi.org/10.1063/1.4998644 (default)
- **auxbasis** (*Optional[BasisInpType]*) – Auxiliary basis for the density fit. If not specified, then it uses `"cc-pvtz-jkfit"`.

get_hamiltonian() → dqc.hamilton.base_hamilton.BaseHamilton

Returns the Hamiltonian that corresponds to the system, i.e. *HamiltonCGTO_PBC*

set_cache(*fname: str*, *paramnames: Optional[List[str]] = None*) → dqc.system.base_system.BaseSystem

Setup the cache of some parameters specified by *paramnames* to be read/written on a file. If the file exists, then the parameters will not be recomputed, but just loaded from the cache instead. Cache is usually used

for repeated calculations where the cached parameters are not changed (e.g. running multiple systems with slightly different environment.)

Parameters

- **fname** (*str*) – The file to store the cache.
- **paramnames** (*list of str or None*) – List of parameter names to be read/write from the cache.

get_orbweight(*polarized: bool = False*) → Union[torch.Tensor, dqc.utils.datastruct.SpinParam[torch.Tensor]]
Returns the atomic orbital weights. If polarized == False, then it returns the total orbital weights. Otherwise, it returns a tuple of orbital weights for spin-up and spin-down.

get_nuclei_energy() → torch.Tensor
Returns the nuclei-nuclei repulsion energy.

setup_grid() → None
Construct the integration grid for the system

get_grid() → dqc.grid.base_grid.BaseGrid
Returns the grid of the system

requires_grid() → bool
True if the system needs the grid to be constructed. Otherwise, returns False

getparamnames(*methodname: str*, *prefix: str = ''*) → List[str]
This method should list tensor names that affect the output of the method with name indicated in `methodname`. If the `methodname` is not on the list in this function, it should raise `KeyError`.

Parameters

- **methodname** (*str*) – The name of the method of the class.
- **prefix** (*str*) – The prefix to be appended in front of the parameters name. This usually contains the dots.

Returns Sequence of name of parameters affecting the output of the method.

Return type Sequence of string

Raises **KeyError** – If the list in this function does not contain `methodname`.

Example

```
>>> class A(xitorch.EditableModule):
...     def __init__(self, a):
...         self.b = a*a
...
...     def mult(self, x):
...         return self.b * x
...
...     def getparamnames(self, methodname, prefix=""):
...         if methodname == "mult":
...             return [prefix+"b"]
...         else:
...             raise KeyError()
```

make_copy(***kwargs*) → dqc.system.sol.Sol
Returns a copy of the system identical to the orginal except for new parameters set in the kwargs.

Parameters ****kwargs** – Must be the same kwargs as Sol.

property atompos
Returns the atom positions as a tensor with shape `(natoms, ndim)`

property atomzs
Returns the tensor containing the atomic number with shape `(natoms,)`

property atommasses
Returns the tensor containing atomic mass with shape `(natoms)` in atomic unit

property spin
Returns the total spin of the system.

property charge
Returns the charge of the system.

property numel
Returns the total number of the electrons in the system.

property efield
Returns the external electric field of the system, or None if there is no electric field.

CHAPTER

FIVE

QCCALC (DQC)

5.1 HF

class dqc.**HF**(*system: dqc.system.base_system.BaseSystem*, *restricted: Optional[bool] = None*, *variational: bool = False*)

Performing Restricted or Unrestricted Kohn-Sham DFT calculation.

Parameters

- **system** (*BaseSystem*) – The system to be calculated.
- **restricted** (*bool or None*) – If True, performing restricted Kohn-Sham DFT. If False, it performs the unrestricted Kohn-Sham DFT. If None, it will choose True if the system is unpolarized and False if it is polarized
- **variational** (*bool*) – If True, then use optimization of the free orbital parameters to find the minimum energy. Otherwise, use self-consistent iterations.

5.2 KS

class dqc.**KS**(*system: dqc.system.base_system.BaseSystem*, *xc: Optional[Union[str, dqc.xc.base_xc.BaseXC]]*, *restricted: Optional[bool] = None*, *variational: bool = False*)

Performing Restricted or Unrestricted Kohn-Sham DFT calculation.

Parameters

- **system** (*BaseSystem*) – The system to be calculated.
- **xc** (*str, BaseXC, or None*) – The exchange-correlation potential and energy to be used. It can accept `None` as an input to represent no xc potential involved.
- **restricted** (*bool or None*) – If True, performing restricted Kohn-Sham DFT. If False, it performs the unrestricted Kohn-Sham DFT. If None, it will choose True if the system is unpolarized and False if it is polarized
- **variational** (*bool*) – If True, then solve the Kohn-Sham equation variationally (i.e. using optimization) instead of using self-consistent iteration. Otherwise, solve it using self-consistent iteration.

CHAPTER

SIX

OTHER APIS (DQC)

6.1 edipole

dqc.**edipole**(*qc: dqc.qccalc.base_qccalc.BaseQCCalc*, *unit: Optional[str] = 'Debye'*) → torch.Tensor
Returns the electric dipole moment of the system, i.e. negative derivative of energy w.r.t. electric field. The dipole is pointing from negative to positive charge.

Parameters

- **qc** (*BaseQCCalc*) – The qc calc object that has been executed.
- **unit** (*str or None*) – The returned dipole unit. If None, returns in atomic unit.

Returns Tensor representing the dipole moment in atomic unit with shape (ndim,)

Return type torch.Tensor

Example

```
import torch
import dqc

dtype = torch.float64
moldesc = "O 0 0 0.2156; H 0 1.4749 -0.8625; H 0 -1.4749 -0.8625"  # in Bohr
efield = torch.zeros(3, dtype=dtype).requires_grad_()  # efield must be specified
mol = dqc.Mol(moldesc=moldesc, basis="3-21G", dtype=dtype, efield=(efield,))
qc = dqc.HF(mol).run()
dip_moment = dqc.edipole(qc, unit="debye")
```

6.2 equadrupole

dqc.**equadrupole**(*qc: dqc.qccalc.base_qccalc.BaseQCCalc*, *unit: Optional[str] = 'Debye*Angst'*) → torch.Tensor
Returns the electric quadrupole moment of the system, i.e. derivative of energy w.r.t. electric field.

Parameters

- **qc** (*BaseQCCalc*) – The qc calc object that has been executed.
- **unit** (*str or None*) – The returned quadrupole unit. If None, returns in atomic unit.

Returns Tensor representing the quadrupole moment in atomic unit in (ndim, ndim)

Return type torch.Tensor

Example

```
import torch
import dqc

dtype = torch.float64
moldesc = "O 0 0 0.2156; H 0 1.4749 -0.8625; H 0 -1.4749 -0.8625"  # in Bohr
efield = torch.zeros(3, dtype=dtype).requires_grad_()  # efield must be specified
grad_efield = torch.zeros((3, 3), dtype=dtype).requires_grad_()  # grad_efield␣
↪must be specified
mol = dqc.Mol(moldesc=moldesc, basis="3-21G", dtype=dtype, efield=(efield, grad_
↪efield))
qc = dqc.HF(mol).run()
equad = dqc.equadrupole(qc)
```

6.3 get_xc

dqc.**get_xc**(*xcstr: str*) → dqc.xc.base_xc.BaseXC

Returns the XC object based on the expression in xcstr.

Parameters **xcstr** (*str*) – The expression of the xc string, e.g. `"lda_x + gga_c_pbe"` where the variable name will be replaced by the LibXC object

Returns XC object based on the given expression

Return type BaseXC

6.4 hessian_pos

dqc.**hessian_pos**(*qc: dqc.qccalc.base_qccalc.BaseQCCalc*, *unit: Optional[str] = None*) → torch.Tensor

Returns the Hessian of energy with respect to atomic positions.

Parameters

- **qc** (*BaseQCCalc*) – Quantum Chemistry calculation that has run.
- **unit** (*str or None*) – The returned unit. If `None`, returns in atomic unit.

Returns Tensor with shape `(natoms * ndim, natoms * ndim)` represents the Hessian of the energy with respect to the atomic position

Return type torch.Tensor

6.5 ir_spectrum

dqc.**ir_spectrum**(*qc: dqc.qccalc.base_qccalc.BaseQCCalc*, *freq_unit: Optional[str] = 'cm^-1'*, *ints_unit: Optional[str] = '(debye/angst)^2/amu'*) → Tuple[torch.Tensor, torch.Tensor]

Calculate the frequency and intensity of the IR vibrational spectra. Unlike `vibration`, this method only returns parts where the frequency is positive.

Parameters

- **qc** (*BaseQCCalc*) – The qc calc object that has been executed.

- **freq_unit** (*str or None*) – The returned unit for the frequency. If `None`, returns in atomic unit.
- **ints_unit** (*str or None*) – The returned unit for the intensity. If `None`, returns in atomic unit.

Returns Tuple of tensors where the first tensor is the frequency in the given unit with shape `(nfreqs,)` sorted from the largest to smallest, and the second tensor is the IR intensity with the same order as the frequency.

Return type Tuple[torch.Tensor, torch.Tensor]

Example

```
import torch
import dqc

dtype = torch.float64
moldesc = "O 0 0 0.2156; H 0 1.4749 -0.8625; H 0 -1.4749 -0.8625"  # in Bohr
efield = torch.zeros(3, dtype=dtype).requires_grad_()  # efield must be specified
mol = dqc.Mol(moldesc=moldesc, basis="3-21G", dtype=dtype, efield=(efield,))
qc = dqc.HF(mol).run()
ir_freq, ir_ints = dqc.ir_spectrum(qc, freq_unit="cm^-1")
```

6.6 is_orb_min

dqc.**is_orb_min**(*qc: dqc.qccalc.base_qccalc.BaseQCCalc*, *threshold: float = - 0.001*) → bool

Check the stability of the SCF if it is the minimum, not a saddle point.

Parameters

- **qc** (*BaseQCCalc*) – The qc calc object that has been executed.
- **threshold** (*float*) – The threshold value of the lowest eigenvalue of the Hessian matrix to be qualified as a positive definite matrix.

Returns `True` if the state is minimum, otherwise returns `False`.

Return type bool

6.7 loadbasis

dqc.**loadbasis**(*cmd: str*, *dtype: torch.dtype = torch.float64*, *device: torch.device = device(type='cpu')*, *requires_grad: bool = False*) → List[dqc.utils.datastruct.CGTOBasis]

Load basis from a file and return the list of CGTOBasis.

Parameters

- **cmd** (*str*) – This can be a file path where the basis is stored or a string in format `"atomz:basis"`, e.g. `"1:6-311++G**"`.
- **dtype** (*torch.dtype*) – Tensor data type for `alphas` and `coeffs` of the GTO basis
- **device** (*torch.device*) – Tensor device for `alphas` and `coeffs`
- **requires_grad** (*bool*) – If `True`, the `alphas` and `coeffs` tensors become differentiable

Returns List of GTO basis loaded from the given file

Return type list of CGTOBasis

6.8 lowest_eival_orb_hessian

dqc.**lowest_eival_orb_hessian**(*qc: dqc.qccalc.base_qccalc.BaseQCCalc*) → torch.Tensor
Get the lowest eigenvalue of the orbital Hessian

Parameters **qc** (*BaseQCCalc*) – The qc calc object that has been executed.

Returns A single-element tensor representing the lowest eigenvalue of the Hessian of energy with respect to orbital parameters. It is useful to check the convergence stability whether it ends up in a ground state or an excited state.

Return type torch.Tensor

6.9 optimal_geometry

dqc.**optimal_geometry**(*qc: dqc.qccalc.base_qccalc.BaseQCCalc*, *length_unit: Optional[str] = None*) → torch.Tensor
Compute the optimal atomic positions of the system.

Parameters

- **qc** (*BaseQCCalc*) – Quantum Chemistry calculation that has run.
- **length_unit** (*str or None*) – The returned unit. If `None`, returns in atomic unit.

Returns Tensor with shape `(natoms, ndim)` represents the position of atoms at the optimal geometry.

Return type torch.Tensor

6.10 parse_moldesc

dqc.**parse_moldesc**(*moldesc: Union[str, Tuple[Union[List[str], List[Union[int, float, torch.Tensor]], torch.Tensor], Union[List[List[float]], numpy.ndarray, torch.Tensor]]]*, *dtype: torch.dtype = torch.float64*, *device: torch.device = device(type='cpu')*) → Tuple[torch.Tensor, torch.Tensor]
Parse the string of molecular descriptor and returns tensors of atomzs and atom positions.

Parameters

- **moldesc** (*str*) – String that describes the system, e.g. `"H -1 0 0; H 1 0 0"` for H2 molecule separated by 2 Bohr.
- **dtype** (*torch.dtype*) – The datatype of the returned atomic positions.
- **device** (*torch.device*) – The device to store the returned tensors.

Returns The first element is the tensor of atomz, and the second element is the tensor of atomic positions.

Return type tuple of 2 tensors

6.11 raman_spectrum

dqc.**raman_spectrum**(*qc: dqc.qccalc.base_qccalc.BaseQCCalc*, *freq_unit: Optional[str] = 'cm^-1'*, *ints_unit: Optional[str] = 'angst^4/amu'*) → Tuple[torch.Tensor, torch.Tensor]

Calculates the frequency, static Raman intensity spectra, and depolarization ratio. Like IR spectrum, this method only returns parts where the frequency is positive.

Parameters

- **qc** (*BaseQCCalc*) – The qc calc object that has been executed.
- **freq_unit** (*str or None*) – The returned unit for the frequency. If None, returns in atomic unit.
- **ints_unit** (*str or None*) – The returned unit for the Raman intensity. If None, returns in atomic unit.

Returns Tuple of tensors where the first tensor is the frequency in the given unit with shape (nfreqs,) sorted from the largest to smallest, and the second tensor is the IR intensity with the same order as the frequency.

Return type Tuple[torch.Tensor, torch.Tensor]

Example

```
import torch
import dqc

dtype = torch.float64
moldesc = "O 0 0 0.2156; H 0 1.4749 -0.8625; H 0 -1.4749 -0.8625"  # in Bohr
efield = torch.zeros(3, dtype=dtype).requires_grad_()  # efield must be specified
mol = dqc.Mol(moldesc=moldesc, basis="3-21G", dtype=dtype, efield=(efield,))
qc = dqc.HF(mol).run()
raman_freq, raman_ints = dqc.raman_spectrum(qc, freq_unit="cm^-1", ints_unit=
↪"angst^4/amu")
```

6.12 vibration

dqc.**vibration**(*qc: dqc.qccalc.base_qccalc.BaseQCCalc*, *freq_unit: Optional[str] = 'cm^-1'*, *length_unit: Optional[str] = None*) → Tuple[torch.Tensor, torch.Tensor]

Calculate the vibration mode of the system based on the Hessian of energy with respect to atomic position.

Parameters

- **qc** (*BaseQCCalc*) – The qc calc object that has been executed.
- **freq_unit** (*str or None*) – The returned unit for the frequency. If None, returns in atomic unit.
- **length_unit** (*str or None*) – The returned unit for the normal mode coordinate. If None, returns in atomic unit

Returns Tuple of tensors where the first tensor is the frequency in atomic unit with shape (natoms * ndim) sorted from the largest to smallest, and the second tensor is the normal coordinate axes in atomic unit with shape (natoms * ndim, natoms * ndim) where each column corresponds to each axis sorted from the largest frequency to smallest frequency.

Return type Tuple[torch.Tensor, torch.Tensor]

CHAPTER

SEVEN

HAMILTON (DQC.HAMILTON)

7.1 HamiltonCGTO

class dqc.hamilton.**HamiltonCGTO**(*atombases: List[dqc.utils.datastruct.AtomCGTOBasis]*, *spherical: bool = True*, *df: Optional[dqc.utils.datastruct.DensityFitInfo] = None*, *efield: Optional[Tuple[torch.Tensor, ...]] = None*, *vext: Optional[torch.Tensor] = None*, *cache: Optional[dqc.utils.cache.Cache] = None*, *orthozer: bool = True*, *aoparamzer: str = 'qr'*)

Hamiltonian object of contracted Gaussian type-orbital. This class orthogonalizes the basis by taking the weighted eigenvectors of the overlap matrix, i.e. the eigenvectors divided by square root of the eigenvalues. The advantage of doing this is making the overlap matrix in Roothan's equation identity and it could handle overcomplete basis.

property nao

Returns the number of atomic orbital basis

property kpts

Returns the list of k-points in the Hamiltonian, raise TypeError if the Hamiltonian does not have k-points. Shape: (nkpts, ndim)

property df

Returns the density fitting object (if any) attached to this Hamiltonian object. If None, returns None

build() → dqc.hamilton.base_hamilton.BaseHamilton

Construct the elements needed for the Hamiltonian. Heavy-lifting operations should be put here.

setup_grid(*grid: dqc.grid.base_grid.BaseGrid*, *xc: Optional[dqc.xc.base_xc.BaseXC] = None*) → None

Setup the basis (with its grad) in the spatial grid and prepare the gradient of atomic orbital according to the ones required by the xc. If xc is not given, then only setup the grid with ao (without any gradients of ao)

get_nuclattr() → xitorch._core.linop.LinearOperator

Returns the LinearOperator of the nuclear Coulomb attraction.

get_kinnucl() → xitorch._core.linop.LinearOperator

Returns the LinearOperator of the one-electron operator (i.e. kinetic and nuclear attraction).

get_overlap() → xitorch._core.linop.LinearOperator

Returns the LinearOperator representing the overlap of the basis.

get_elrep(*dm: torch.Tensor*) → xitorch._core.linop.LinearOperator

Obtains the LinearOperator of the Coulomb electron repulsion operator. Known as the J-matrix.

get_exchange(*dm: torch.Tensor*) → xitorch._core.linop.LinearOperator

get_exchange(*dm: dqc.utils.datastruct.SpinParam[torch.Tensor]*) → dqc.utils.datastruct.SpinParam[xitorch._core.linop.LinearOperator]
Obtains the LinearOperator of the exchange operator. It is -0.5 * K where K is the K matrix obtained from 2-electron integral.

get_vext(*vext: torch.Tensor*) → xitorch._core.linop.LinearOperator
Returns a LinearOperator of the external potential in the grid.

$$\mathbf{V}_{ij} = \int b_i(\mathbf{r}) V(\mathbf{r}) b_j(\mathbf{r})\, d\mathbf{r}$$

get_vxc(*dm: dqc.utils.datastruct.SpinParam[torch.Tensor]*) → dqc.utils.datastruct.SpinParam[xitorch._core.linop.LinearOper
get_vxc(*dm: torch.Tensor*) → xitorch._core.linop.LinearOperator
Returns a LinearOperator for the exchange-correlation potential.

ao_orb2dm(*orb: torch.Tensor*, *orb_weight: torch.Tensor*) → torch.Tensor
Convert the atomic orbital to the density matrix.

aodm2dens(*dm: torch.Tensor*, *xyz: torch.Tensor*) → torch.Tensor
Get the density value in the Cartesian coordinate.

get_e_hcore(*dm: torch.Tensor*) → torch.Tensor
Get the energy from the one-electron Hamiltonian. The input is total density matrix.

get_e_elrep(*dm: torch.Tensor*) → torch.Tensor
Get the energy from the electron repulsion. The input is total density matrix.

get_e_exchange(*dm: Union[torch.Tensor, dqc.utils.datastruct.SpinParam[torch.Tensor]]*) → torch.Tensor
Get the energy from the exact exchange.

get_e_xc(*dm: Union[torch.Tensor, dqc.utils.datastruct.SpinParam[torch.Tensor]]*) → torch.Tensor
Returns the exchange-correlation energy using the xc object given in `.setup_grid()`

ao_orb_params2dm(*ao_orb_params: torch.Tensor*, *ao_orb_coeffs: torch.Tensor*, *orb_weight: torch.Tensor*, *with_penalty: None*) → torch.Tensor
ao_orb_params2dm(*ao_orb_params: torch.Tensor*, *ao_orb_coeffs: torch.Tensor*, *orb_weight: torch.Tensor*, *with_penalty: float*) → torch.Tensor
Convert the atomic orbital free parameters (parametrized in such a way so it is not bounded) to the density matrix.

Parameters

- **ao_orb_params** (*torch.Tensor*) – The tensor that parametrized atomic orbital in an unbounded space.
- **ao_orb_coeffs** (*torch.Tensor*) – The tensor that helps `ao_orb_params` in describing the orbital. The difference with `ao_orb_params` is that `ao_orb_coeffs` is not differentiable and not to be optimized in variational method.
- **orb_weight** (*torch.Tensor*) – The orbital weights.
- **with_penalty** (*float or None*) – If a float, it returns a tuple of tensors where the first element is `dm`, and the second element is the penalty multiplied by the penalty weights. The penalty is to compensate the overparameterization of `ao_orb_params`, stabilizing the Hessian for gradient calculation.

Returns The density matrix from the orbital parameters and (if `with_penalty`) the penalty of the overparameterization of `ao_orb_params`.

Return type torch.Tensor or tuple of torch.Tensor

Notes

- The penalty should be 0 if `ao_orb_params` is from `dm2ao_orb_params`.
- The density matrix should be recoverable when put through `dm2ao_orb_params` and `ao_orb_params2dm`.

dm2ao_orb_params(*dm: torch.Tensor*, *norb: int*) → Tuple[torch.Tensor, torch.Tensor]
Convert from the density matrix to the orbital parameters. The map is not one-to-one, but instead one-to-many where there might be more than one orbital parameters to describe the same density matrix. For restricted systems, only one of the `dm` (`dm.u` or `dm.d`) is sufficient.

Parameters

- **dm** (*torch.Tensor*) – The density matrix.
- **norb** (*int*) – The number of orbitals for the system.

Returns The atomic orbital parameters for the first returned value and the atomic orbital coefficients for the second value.

Return type tuple of 2 torch.Tensor

getparamnames(*methodname: str*, *prefix: str = ''*) → List[str]
Return the paramnames

7.2 HamiltonCGTO_PBC

class dqc.hamilton.**HamiltonCGTO_PBC**(*atombases: List[dqc.utils.datastruct.AtomCGTOBasis]*, *latt: dqc.hamilton.intor.lattice.Lattice*, ***, *kpts: Optional[torch.Tensor] = None*, *wkpts: Optional[torch.Tensor] = None*, *spherical: bool = True*, *df: Optional[dqc.utils.datastruct.DensityFitInfo] = None*, *lattsum_opt: Optional[Union[dqc.hamilton.intor.pbcintor.PBCIntOption, Dict]] = None*, *cache: Optional[dqc.utils.cache.Cache] = None*)
Hamiltonian with contracted Gaussian type orbitals in a periodic boundary condition systems. The calculation of Hamiltonian components follow the reference: Sun, et al., J. Chem. Phys. 147, 164119 (2017) https://doi.org/10.1063/1.4998644

property nao
Returns the number of atomic orbital basis

property kpts
Returns the list of k-points in the Hamiltonian, raise TypeError if the Hamiltonian does not have k-points. Shape: (nkpts, ndim)

property df
Returns the density fitting object (if any) attached to this Hamiltonian object. If None, returns None

build() → dqc.hamilton.base_hamilton.BaseHamilton
Construct the elements needed for the Hamiltonian. Heavy-lifting operations should be put here.

setup_grid(*grid: dqc.grid.base_grid.BaseGrid*, *xc: Optional[dqc.xc.base_xc.BaseXC] = None*) → None
Setup the basis (with its grad) in the spatial grid and prepare the gradient of atomic orbital according to the ones required by the xc. If xc is not given, then only setup the grid with ao (without any gradients of ao)

get_nuclattr() → xitorch._core.linop.LinearOperator
Returns the LinearOperator of the nuclear Coulomb attraction.

get_kinnucl() → xitorch._core.linop.LinearOperator
Returns the LinearOperator of the one-electron operator (i.e. kinetic and nuclear attraction).

get_overlap() → xitorch._core.linop.LinearOperator
Returns the LinearOperator representing the overlap of the basis.

get_elrep(*dm: torch.Tensor*) → xitorch._core.linop.LinearOperator
Obtains the LinearOperator of the Coulomb electron repulsion operator. Known as the J-matrix.

get_exchange(*dm: torch.Tensor*) → xitorch._core.linop.LinearOperator
get_exchange(*dm: dqc.utils.datastruct.SpinParam[torch.Tensor]*) → dqc.utils.datastruct.SpinParam[xitorch._core.linop.LinearOperator]
Obtains the LinearOperator of the exchange operator. It is -0.5 * K where K is the K matrix obtained from 2-electron integral.

get_vext(*vext: torch.Tensor*) → xitorch._core.linop.LinearOperator
Returns a LinearOperator of the external potential in the grid.

$$\mathbf{V}_{ij} = \int b_i(\mathbf{r}) V(\mathbf{r}) b_j(\mathbf{r})\, d\mathbf{r}$$

get_vxc(*dm: dqc.utils.datastruct.SpinParam[torch.Tensor]*) → dqc.utils.datastruct.SpinParam[xitorch._core.linop.LinearOper
get_vxc(*dm: torch.Tensor*) → xitorch._core.linop.LinearOperator
Returns a LinearOperator for the exchange-correlation potential.

ao_orb2dm(*orb: torch.Tensor*, *orb_weight: torch.Tensor*) → torch.Tensor
Convert the atomic orbital to the density matrix.

aodm2dens(*dm: torch.Tensor*, *xyz: torch.Tensor*) → torch.Tensor
Get the density value in the Cartesian coordinate.

get_e_hcore(*dm: torch.Tensor*) → torch.Tensor
Get the energy from the one-electron Hamiltonian. The input is total density matrix.

get_e_elrep(*dm: torch.Tensor*) → torch.Tensor
Get the energy from the electron repulsion. The input is total density matrix.

get_e_exchange(*dm: Union[torch.Tensor, dqc.utils.datastruct.SpinParam[torch.Tensor]]*) → torch.Tensor
Get the energy from the exact exchange.

get_e_xc(*dm: Union[torch.Tensor, dqc.utils.datastruct.SpinParam[torch.Tensor]]*) → torch.Tensor
Returns the exchange-correlation energy using the xc object given in `.setup_grid()`

getparamnames(*methodname: str*, *prefix: str = ''*) → List[str]
Return the paramnames

CHAPTER

EIGHT

XC (DQC.XC)

8.1 CustomXC

class dqc.xc.**CustomXC**

Base class of custom xc functional.

abstract property family

Returns 1 for LDA, 2 for GGA, and 4 for Meta-GGA.

abstract get_edensityxc(*densinfo: Union[dqc.utils.datastruct.ValGrad, dqc.utils.datastruct.SpinParam[dqc.utils.datastruct.ValGrad]]*) → torch.Tensor

Returns the xc energy density (energy per unit volume)

getparamnames(*methodname: str = ''*, *prefix: str = ''*) → List[str]

This method should list tensor names that affect the output of the method with name indicated in `methodname`. If the `methodname` is not on the list in this function, it should raise `KeyError`.

Parameters

- **methodname** (*str*) – The name of the method of the class.
- **prefix** (*str*) – The prefix to be appended in front of the parameters name. This usually contains the dots.

Returns Sequence of name of parameters affecting the output of the method.

Return type Sequence of string

Raises **KeyError** – If the list in this function does not contain `methodname`.

Example

```
>>> class A(xitorch.EditableModule):
...     def __init__(self, a):
...         self.b = a*a
...
...     def mult(self, x):
...         return self.b * x
...
...     def getparamnames(self, methodname, prefix=""):
...         if methodname == "mult":
...             return [prefix+"b"]
...         else:
...             raise KeyError()
```

CHAPTER

NINE

UTILITIES (DQC.UTILS)

9.1 SpinParam

class dqc.utils.**SpinParam**(**args*, ***kwds*)

Data structure to store different values for spin-up and spin-down electrons.

u

The parameters that corresponds to the spin-up electrons.

Type any type

d

The parameters that corresponds to the spin-down electrons.

Type any type

Example

```
import torch
import dqc.utils
dens_u = torch.ones(1)
dens_d = torch.zeros(1)
sp = dqc.utils.SpinParam(u=dens_u, d=dens_d)
print(sp.u)
```

```
tensor([1.])
```

9.2 ValGrad

class dqc.utils.**ValGrad**(*value: torch.Tensor*, *grad: Optional[torch.Tensor] = None*, *lapl: Optional[torch.Tensor] = None*, *kin: Optional[torch.Tensor] = None*)

Data structure that contains local information about density profiles.

value

Tensors containing the value of the local information.

Type torch.Tensor

grad

If tensor, it represents the gradient of the local information with shape `(..., 3)` where `...` should be the same shape as `value`.

Type torch.Tensor or None

lapl
If tensor, represents the laplacian value of the local information. It should have the same shape as `value`.

Type torch.Tensor or None

kin
If tensor, represents the local kinetic energy density. It should have the same shape as `value`.

Type torch.Tensor or None

9.3 convert_length

dqc.utils.**convert_length**(*a: torch.Tensor*, *from_unit: Optional[str] = None*, *to_unit: Optional[str] = None*) → torch.Tensor

Convert the length from a unit to another unit. Available units are (case-insensitive): `['angst', 'angstrom', 'm', 'cm']`

Parameters

- **a** (*torch.Tensor*) – The tensor to be converter.
- **from_unit** (*str or None*) – The unit of `a`. If `None`, it is assumed to be in atomic unit.
- **to_unit** (*str or None*) – The unit for `a` to be converted to. If `None`, it is assumed to be converted to the atomic unit.

Returns The tensor in the new unit.

Return type torch.Tensor

9.4 convert_time

dqc.utils.**convert_time**(*a: torch.Tensor*, *from_unit: Optional[str] = None*, *to_unit: Optional[str] = None*) → torch.Tensor

Convert the time from a unit to another unit. Available units are (case-insensitive): `['s', 'us', 'ns', 'fs']`

Parameters

- **a** (*torch.Tensor*) – The tensor to be converter.
- **from_unit** (*str or None*) – The unit of `a`. If `None`, it is assumed to be in atomic unit.
- **to_unit** (*str or None*) – The unit for `a` to be converted to. If `None`, it is assumed to be converted to the atomic unit.

Returns The tensor in the new unit.

Return type torch.Tensor

9.5 convert_freq

dqc.utils.**convert_freq**(*a: torch.Tensor*, *from_unit: Optional[str] = None*, *to_unit: Optional[str] = None*) → torch.Tensor

Convert the frequency from a unit to another unit. Available units are (case-insensitive): `['cm-1', 'cm^-1', 'hz', 'khz', 'mhz', 'ghz', 'thz']`

Parameters

- **a** (*torch.Tensor*) – The tensor to be converter.
- **from_unit** (*str or None*) – The unit of `a`. If `None`, it is assumed to be in atomic unit.
- **to_unit** (*str or None*) – The unit for `a` to be converted to. If `None`, it is assumed to be converted to the atomic unit.

Returns The tensor in the new unit.

Return type torch.Tensor

9.6 convert_ir_ints

dqc.utils.**convert_ir_ints**(*a: torch.Tensor*, *from_unit: Optional[str] = None*, *to_unit: Optional[str] = None*) → torch.Tensor

Convert the IR intensity from a unit to another unit. Available units are (case-insensitive): `['(debye/angst)^2/amu', 'km/mol']`

Parameters

- **a** (*torch.Tensor*) – The tensor to be converter.
- **from_unit** (*str or None*) – The unit of `a`. If `None`, it is assumed to be in atomic unit.
- **to_unit** (*str or None*) – The unit for `a` to be converted to. If `None`, it is assumed to be converted to the atomic unit.

Returns The tensor in the new unit.

Return type torch.Tensor

9.7 convert_raman_ints

dqc.utils.**convert_raman_ints**(*a: torch.Tensor*, *from_unit: Optional[str] = None*, *to_unit: Optional[str] = None*) → torch.Tensor

Convert the Raman intensity from a unit to another unit. Available units are (case-insensitive): `['angst^4/amu']`

Parameters

- **a** (*torch.Tensor*) – The tensor to be converter.
- **from_unit** (*str or None*) – The unit of `a`. If `None`, it is assumed to be in atomic unit.
- **to_unit** (*str or None*) – The unit for `a` to be converted to. If `None`, it is assumed to be converted to the atomic unit.

Returns The tensor in the new unit.

Return type torch.Tensor

9.8 convert_edipole

dqc.utils.**convert_edipole**(*a: torch.Tensor*, *from_unit: Optional[str] = None*, *to_unit: Optional[str] = None*) → torch.Tensor

Convert the electric dipole from a unit to another unit. Available units are (case-insensitive): `['d', 'debye', 'c*m']`

Parameters

- **a** (*torch.Tensor*) – The tensor to be converter.
- **from_unit** (*str or None*) – The unit of `a`. If `None`, it is assumed to be in atomic unit.
- **to_unit** (*str or None*) – The unit for `a` to be converted to. If `None`, it is assumed to be converted to the atomic unit.

Returns The tensor in the new unit.

Return type torch.Tensor

9.9 convert_equadrupole

dqc.utils.**convert_equadrupole**(*a: torch.Tensor*, *from_unit: Optional[str] = None*, *to_unit: Optional[str] = None*) → torch.Tensor

Convert the electric quadrupole from a unit to another unit. Available units are (case-insensitive): `['debye*angst']`

Parameters

- **a** (*torch.Tensor*) – The tensor to be converter.
- **from_unit** (*str or None*) – The unit of `a`. If `None`, it is assumed to be in atomic unit.
- **to_unit** (*str or None*) – The unit for `a` to be converted to. If `None`, it is assumed to be converted to the atomic unit.

Returns The tensor in the new unit.

Return type torch.Tensor

CHAPTER

TEN

RELEASE NOTES

10.1 Release notes v0.2.0 (upcoming)

10.1.1 New features

- Options in `Mol` to use orthogonalized basis or non-orthogonalized basis.

10.1.2 Bug fixes

- Fixing bug in evaluating the density profile using orthogonalized basis.

10.2 Release notes v0.1.0

First release of DQC:

- Hartree-Fock (HF) calculation on isolated molecules (restricted + unrestricted) using linear combinations of contracted gaussian orbitals
- Kohn-Sham DFT (KS) calculation on isolated molecules (restricted + unrestricted) using linear combinations of contracted gaussian orbitals
- Various perturbation properties: Raman activity, IR intensity, vibrational frequency and normal modes, and SCF convergence stability check
- (experimental) Direct minimization of orbitals to solve HF and KS-DFT equation
- Fractional Z systems
- Differentiability with respect to: atomic numbers, electron occupation numbers, atomic positions, basis contracted coefficients and exponential factors, xc functional parameters, external potential.

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- search

INDEX

H

I

K

L

M

N

O

P

R

S

U

V