QMCTorch Documentation

Release 0.1.0

Nicolas Renaud

Apr 25, 2023

QMCTorch

CHAPTER 1

Installation

1.1 Via Python Package

The latest release of QMCTorch can be installed usint the pypi package manager with :

```
pip install qmctorch
```

If you are planning to only use QMCTorch this is the quickest way to obtain the code

1.2 Via GitHub

For user who would like to contribute, the code is hosted on GitHub (https://github.com/NLESC-JCER/QMCTorch)

To install the code

- clone the repository `git clone https://github.com/NLESC-JCER/QMCTorch.git`
- go there `cd QMCTorch`
- install the module `pip insall -e ./`

You can then test the installation :

- `cd test`
- `pytest`

CHAPTER 2

Quantum Monte Carlo: a 1 min introduction

Quantum Monte Carlo simulations rely on the variational principle:

$$E = \frac{\int \Psi_\theta^*(R)\ H\ \Psi_\theta(R) dR}{\int |\Psi_\theta(R)|^2} \geq E_0$$

where $\Psi_\theta(R)$ is the wave function of the system computed for the atomic and electronic positions R, and with variational parameters θ, H is the Hamiltonian of the system given by:

$$H = -\frac{1}{2}\sum_i \Delta_i + \sum_{i>j} \frac{1}{|r_i - r_j|} - \sum_{i\alpha} \frac{Z_\alpha}{|r_i - R_\alpha|} - \sum_{\alpha>\beta} \frac{Z_\alpha Z_\beta}{|R_\alpha - R_\beta|}$$

where Δ_i is the Laplacian w.r.t the i-th electron, r_i is the position of the i-th electron, R_α the position of the α-th atom and Z_α its atomic number. QMC simulations express this integral as:

$$E = \int \rho(R) E_L(R) dR \geq E_0$$

with:

$$\rho(R) = \frac{|\Psi_\theta(R)|^2}{\int |\Psi_\theta(R)|^2 dR}$$

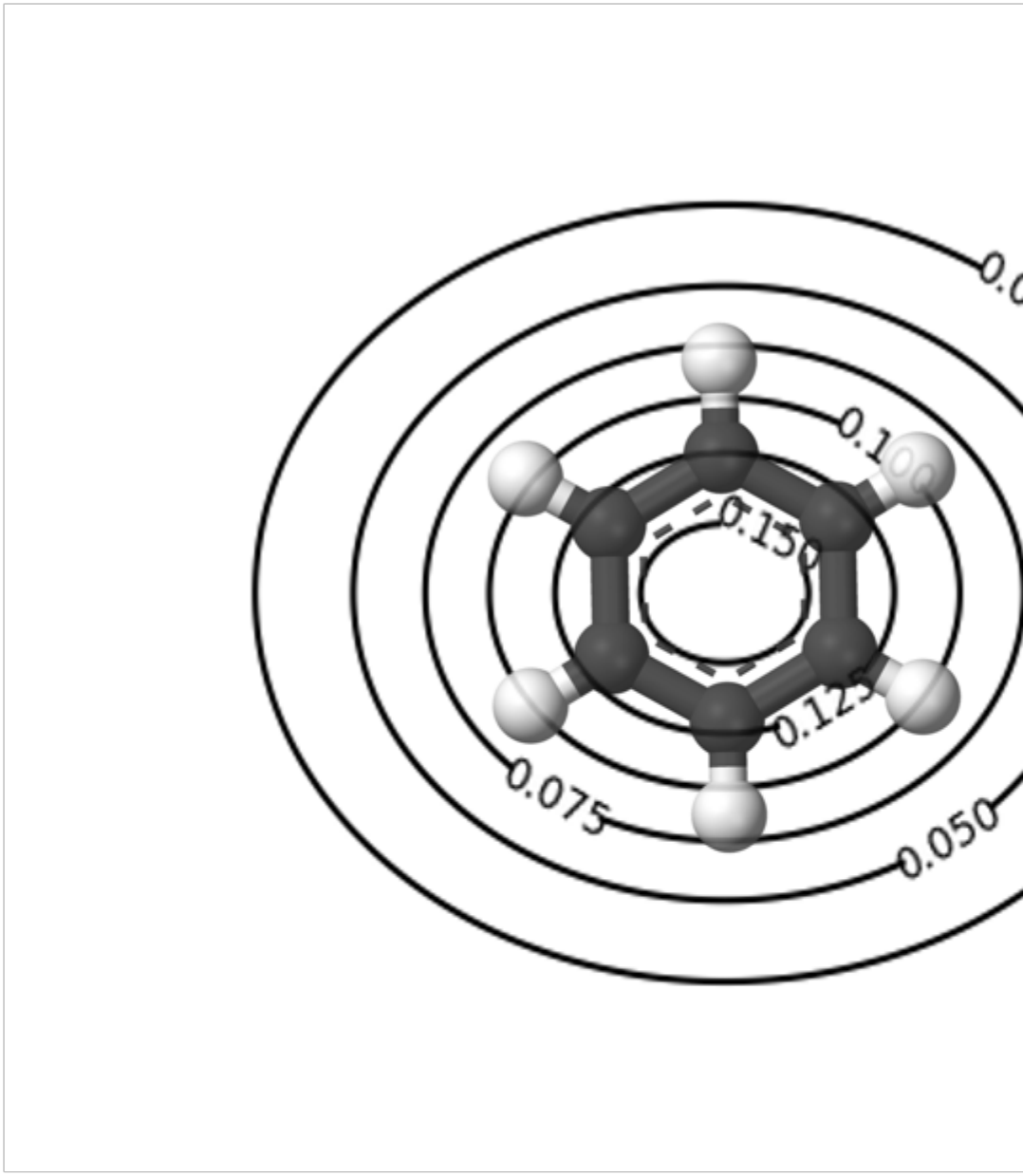
reprensent the denisty associated with the wave function, and:

$$E_L(R) = \frac{H\Psi_\theta(R)}{\Psi_\theta(R)}$$

are the so called local energies of the system. QMC simulation then approximated the total energy as:

$$E \approx \frac{1}{M}\sum_i^M \frac{H\Psi_\theta(R_i)}{\Psi_\theta(R_i)}$$

where R_i are samples of the density ρ for example obtained via Metropolis Hasting sampling. QMC simulations rely then on the optimization of the variational parameters of the wave function, θ, to minimize the value of the total energy of the system.



Define Wave Fun
Functional For

CHAPTER 3

Wave Function ansatz in QMCTorch

QMCTorch allows to epxress the wave function ususally used by QMC practitioner as neural network. The most generic architecture of the neural network used by the code is:

Starting from the electronic and atomic coordinates, the first layer on the bottom computes the electron-electron and electron-atoms distances. These distances are used in a Jastrow layer that computes the Jastrow facrtor. Users can freely define Jastro kernels to define the exact form the Jastrow factor.

In parallel the electronic coordinates are first transformed through a backflow transformation. Users can here as well specify the kernel of the backflow transformation. The resulting new coordinates are used to evaluate the atomic orbitals of the systems. The basis set information of these orbitals are extracted from the SCF calculation performed with `pyscf` or `ADF`. These atomic orbital values are then transformed to molecular orbital values through the next layer. The coefficients of the molecular orbitals are also extracted fron the SCF calculations. Then a Slater determinant layer extract the different determinants contained in the wave function. Users can there as well specify wich determinants they require. The weighted sum of the determinants is then computed and finally muliplied with the value of the Jastrow factor.

Different wave function forms have been implemented to easily create and use wave function ansatz. These different functional forms differ mainly by the Jastrow factor they use and the presence of backflow transformation or not.

3.1 Two-body Jastrow factors

In its simplest form the Jastrow factor only depends on the electron-electron distances. This means that the Jastrow layer only has a single kernel function :math: *K_{ee}*. This Jastrow factor can be applied globally, or different Jastrow factors can be applied to individual orbitals. In addition a Backflow transformation can be added or not to the definition of the wave function. We therefore have the following wave function available:

- `SlaterJastrow`: A simple wave function containing an electron-electron Jastrow factor and a sum of Slater determinants
- `SlaterOrbitalDependentJastrow`: A `SlaterJastrow` for but each molecular orbitals has its own Jastrow factor
- `SlaterJastrowBackflow`: A `SlaterJastrow` wave function with backflow transformation for the electrons

3.1.1 Slater Jastrow Wave Function

The simplest wave function implemented in *QMCTorch* is a Slater Jastrow form. Through a series of transformations the Slater Jastrow function computes:

$$\Psi(R) = J(R) \sum_n c_n D_n^{\uparrow} D_n^{\downarrow}$$

The term *J(R)* is the so called Jastrow factor that captures the electronic correlation. By default, the Jastrow factor is given by :

$$J(R) = \exp\left(\sum_{i<j} \text{Kernel}(r_{ij})\right)$$

where the sum runs over all the electron pairs and where the kernel function defines the action of the Jastrow factor. A common expression for the kernel (and the default option for QMCTorch) is the Pade-Jastrow form given by:

$$\text{Kernel}(r_{ij}) = \frac{\omega_0 r_{ij}}{1 + \omega r_{ij}}$$

where ω_0 is a fixed coefficient equals to 0.25(0.5) for antiparallel(parallel) electron spins and ω a variational parameter.

The determinantal parts in the expression of Ψ are given by the spin-up and spin-down slater determinants e.g. :

$$D_n^{\uparrow} = \frac{1}{\sqrt{N}} \begin{vmatrix} & \\ & \phi_j(r_i) \\ & \end{vmatrix}$$

A `SlaterJastrow` wave function can instantiated following :

```
>>> wf = SlaterJastrow(mol, configs='single_double(2,2)', jastrow_
↪kernel=PadeJastrowKernel)
```

The `SlaterJastrow` takes as first mandiatory argument a `Molecule` instance. The Slater determinants required in the calculation are specified with the `configs` arguments which can take the following values :

- `configs='ground_state'` : only the ground state SD
- `configs='cas(n,m)'` : complete active space using n electron and m orbitals
- `configs='single(n,m)'` : only single excitation using n electron and m orbitals
- `configs='single_double(n,m)'` : only single/double excitation using n electron and m orbitals

Finally the kernel function of the Jastrow factor can be specifed using the `jastrow_kernel` The `SlaterJastrow` class accepts other initialisation arguments to fine tune some advanced settings. The default values of these arguments are adequeate for most cases.

3.1.2 Orbital dependent Slater Jastrow Wave Function

A slight modification of the the Slater Jastrow is obtained by making the the Jastrow factor can be made orbital dependent. This is implemented in the `SlaterOrbitalDependentJastrow` that can be instantiated as:

```
>>> from qmctorch.wavefunction import SlaterOrbitalDependentJastrow
>>> from qmctorch.wavefunction.jastrows.elec_elec.kernels import PadeJastrowKernel
>>> wf = SlaterOrbitalDependentJastrow(mol, configs='single_double(2,4)'
>>>                                    jastrow_kernel=PadeJastrowKernel)
```

3.1.3 Slater Jastrow Backflow Wave Function

The Slater Jastrow Backflow wave function builds on the the Slater Jastrow wavefunction but adds a backflow transformation to the electronic positions. Following this transformation, each electron becomes a quasi-particle whose position depends on all electronic positions. The backflow transformation is given by :

$$q(x_i) = x_i + \sum_{j \neq i} \text{Kernel}(r_{ij})(x_i - x_j)$$

The kernel of the transformation can be any function that depends on the distance between two electrons. A popular kernel is simply the inverse function :

$$\text{Kernel}(r_{ij}) = \frac{\omega}{r_{ij}}$$

and is the default value in QMCTorch. However any other kernel function can be implemented and used in the code.

The wave function is then constructed as :

$$\Psi(R) = J(R) \sum_n c_n D_n^{\uparrow}(Q) D_n^{\downarrow}(Q)$$

The Jastrow factor is still computed using the original positions of the electrons while the determinant part uses the backflow transformed positions. One can define such wave function with:

```
>>> from qmctorch.wavefunction import SlaterJastrowBackFlow
>>> from qmctorch.wavefunction.orbitals.backflow.kernels import BackFlowKernelInverse
>>> from qmctorch.wavefunction.jastrows.elec_elec.kernels import PadeJastrowKernel
>>>
>>> wf = SlaterJastrowBackFlow(mol,
>>>                            configs='single_double(2,2)',
>>>                            jastrow_kernel=PadeJastrowKernel,
>>>                            backflow_kernel=BackFlowKernelInverse)
```

Compared to the `SlaterJastrow` wave function, the kernel of the backflow transformation must be specified. By default the inverse kernel will be used.

Orbital Dependent Backflow Transformation

The backflow transformation can be different for each atomic orbitals.

$$q^{\alpha}(x_i) = x_i + \sum_{j \neq i} \text{Kernel}^{\alpha}(r_{ij})(x_i - x_j)$$

where each orbital has its dedicated backflow kernel. This provides much more flexibility when optimizing the wave function.

This wave function can be used with

```
>>> from qmctorch.wavefunction import SlaterJastrowBackFlow
>>> from qmctorch.wavefunction.orbitals.backflow.kernels import BackFlowKernelInverse
>>> from qmctorch.wavefunction.jastrows.elec_elec.kernels import PadeJastrowKernel
>>>
>>> wf = SlaterJastrowBackFlow(mol,
>>>                            configs='single_double(2,2)',
>>>                            jastrow_kernel=PadeJastrowKernel,
>>>                            orbital_dependent_backflow=True,
>>>                            backflow_kernel=BackFlowKernelInverse)
```

3.2 Many-Body Jastrow factors

Jastrow factors can also depends on the electron-nuclei distances and the many body terms involving two electrons and one nuclei. In that case the Jastrow factor depends on all the kernel function represented in the figure above. A backflow transformation can also be added to the definition of the wave function. As a result we have the following wave function forms available.

- `SlaterCombinedJastrow`: A wave function that contains a many body Jastrow factor and a sum of Slater determinants with backflow transformation for the electrons
- `SlaterCombinedJastrowBackflow`: A `SlaterCombinedJastrow` wave function with a backflow transformation

3.2.1 Many-Body Jastrow Wave Function

The Jastrow factor combines here multiple terms that represent electron-electron, electron-nuclei and electron-electron-nuclei terms.

$$J(R_{at}, r) = \exp\left(\sum_{i<j} K_{ee}(r_i, r_j) + \sum_{i,\alpha} K_{en}(R_\alpha, r_i) + \sum_{i<j,\alpha} K_{een}(R_\alpha, r_i, r_j)\right)$$

```
>>> from qmctorch.wavefunction import SlaterCombinedJastrow
>>> from qmctorch.wavefunction.jastrows.elec_elec.kernels import PadeJastrowKernel as␣
↪PadeJastrowElecElec
>>> from qmctorch.wavefunction.jastrows.elec_nuclei.kernels import PadeJastrowKernel␣
↪as PadeJastrowKernelElecNuc
>>> from qmctorch.wavefunction.jastrows.elec_elec_nuclei.kernels import␣
↪BoysHandyJastrowKernel
>>>
>>> wf = SlaterCombinedJastrow(mol,
>>>                            configs='single_double(2,2)',
>>>                            jastrow_kernel={
>>>                                 'ee': PadeJastrowKernelElecElec,
>>>                                 'en': PadeJastrowKernelElecNuc,
>>>                                 'een': BoysHandyJastrowKernel})
```

3.2.2 Many-Body Jastrow Wave Function with backflow transformation

A backflow transformation can be used together with the many body Jastrow

```
>>> from qmctorch.wavefunction import SlaterCombinedJastrowBackflow
>>> from qmctorch.wavefunction.jastrows.elec_elec.kernels.pade_jastrow_kernel import␣
↪PadeJastrowKernel as PadeJastrowElecElec
>>> from qmctorch.wavefunction.jastrows.elec_nuclei.kernels.pade_jastrow_kernel␣
↪import PadeJastrowKernel as PadeJastrowKernelElecNuc
>>> from qmctorch.wavefunction.jastrows.elec_elec_nuclei.kernels.boys_handy_jastrow_
↪kernel import BoysHandyJastrowKernel
>>>
>>> wf = SlaterCombinedJastrowBackflow(mol,
>>>                            configs='single_double(2,2)',
>>>                            jastrow_kernel={
>>>                                 'ee': PadeJastrowKernelElecElec,
>>>                                 'en': PadeJastrowKernelElecNuc,
>>>                                 'een': BoysHandyJastrowKernel},
>>>                             backflow_kernel=BackFlowKernelInverse)
```

CHAPTER 4

Exploring the results with h5web

The results and input of any the calculation performed by QMCTorch is stored in a dedicated HDF5 File that can be explored using *h5web* (https://h5web.panosc.eu/) A dedicated VS Code extension also allows to browse hdf5 files in your editor

4.1 SCF calculation

The result of any SCF calculation generates an hdf5 file named by default

```
<molecule_name>_<calculator>_<basis>.hdf5
```

Another name can be specified via the Molecule argument. This file contains all the data of the Molecule instance, in particular the basis set information, used in the calculation. By browsing the file using h5x you can retreive any information needed. This file is also reused if possible to avoid computing the SCF of a previously studied molecular system.

4.2 QMCTorch calculation

The results of QMCTorch calculation are also stored in a hdf5 file named

```
<molecule_name>_<calculator>_<basis>_QMCTorch.hdf5
```

Another name can be specified via the Solver argument. This file contains all the information about the wavefunction, the sampler the solver and the results of all the calculations performed in the script. Context menus have been incorporate in order to plot the results directly from the hdf5 file.

CHAPTER 5

Create a molecule

In this tutorial we present how to create a molecule and run the SCF calculation. First, the `Molecule` class must be imported :

[1]:
```
from qmctorch.scf import Molecule
```

```
INFO:QMCTorch|  _____     __  _________________              _
INFO:QMCTorch| / __ \   /  |/  / ____/_  __/__  _________/ /
INFO:QMCTorch|/ /_/ / / / /|_/ / /__   / / / _ \/ __/ __/ _ \
INFO:QMCTorch|\___\_\/_/  /_/\___/ /_/  \___/_/  \__/_//_/
```

This class can interface with `pyscf` and `ADF` to perform SCF calculations. Of course both software use different types of atomic orbitals, respectively Gaussian type orbitals for `pyscf` and Slater type orbitals for `ADF`.

5.1 Geometry of the molecule

The geometry of the molecule can be specified through the `atom` keyword of the `Molecule` class. The units of the positions, `bohr` or `angs` (default is 'bohr') can also be specified via the `unit` keyword argument. The geometry can be passed as a single string

[2]:
```
mol = Molecule(atom = 'H 0. 0. 0; H 0. 0. 1.', unit='bohr')
```

```
INFO:QMCTorch|
INFO:QMCTorch| SCF Calculation
INFO:QMCTorch|  Running scf  calculation
converged SCF energy = -1.06599946214331
INFO:QMCTorch|  Molecule name       : H2
INFO:QMCTorch|  Number of electrons : 2
INFO:QMCTorch|  SCF calculator      : pyscf
INFO:QMCTorch|  Basis set           : sto-3g
INFO:QMCTorch|  SCF                 : HF
INFO:QMCTorch|  Number of AOs       : 2
INFO:QMCTorch|  Number of MOs       : 2
INFO:QMCTorch|  SCF Energy          : -1.066 Hartree
```

```
[2]: <qmctorch.scf.molecule.Molecule at 0x7f429ed23910>
```

or via an XYZ file containing the geomtry of the molecular structure. Note that by default `QMCTorch` will try to reuse previous calculations that might be stored in an hdf5 file. To redo the scf calculation we can use the `redo_scf=True` argument.

```
[7]: mol = Molecule(atom='h2.xyz', unit='bohr', redo_scf=True)
```

```
INFO:QMCTorch|
INFO:QMCTorch| SCF Calculation
INFO:QMCTorch|  Removing H2_pyscf_sto-3g.hdf5 and redo SCF calculations
INFO:QMCTorch|  Running scf  calculation
converged SCF energy = -1.06599946214331
INFO:QMCTorch|  Molecule name       : H2
INFO:QMCTorch|  Number of electrons : 2
INFO:QMCTorch|  SCF calculator      : pyscf
INFO:QMCTorch|  Basis set           : sto-3g
INFO:QMCTorch|  SCF                 : HF
INFO:QMCTorch|  Number of AOs       : 2
INFO:QMCTorch|  Number of MOs       : 2
INFO:QMCTorch|  SCF Energy          : -1.066 Hartree
```

5.2 SCF calculations

As mentionned above `QMCTorch` can use `pyscf` or `ADF` to perform SCF calculation on the molecular structure. At the moment only Hartree-Fock calculations are supported but DFT calculations will be implemented later. We present here how to perform these SCF calculations.

5.2.1 Gaussian orbitals with pyscf

As seen above the code use by default `pyscf` to compute the atomic and molecular orbitals of the system using a `sto-3g` basis set. The default behavior is equivlament to setting `calculator=pyscf` and `basis='sto-3g'`. Let's switch to another basis, e.g. `sto-6g`

```
[10]: mol = Molecule(atom='H 0. 0. 0; H 0. 0. 1.', unit='bohr', calculator='pyscf', basis=
      ↪'sto-6g', redo_scf=True)
```

```
INFO:QMCTorch|
INFO:QMCTorch| SCF Calculation
INFO:QMCTorch|  Running scf  calculation
converged SCF energy = -1.07589040772972
INFO:QMCTorch|  Molecule name       : H2
INFO:QMCTorch|  Number of electrons : 2
INFO:QMCTorch|  SCF calculator      : pyscf
INFO:QMCTorch|  Basis set           : sto-6g
INFO:QMCTorch|  SCF                 : HF
INFO:QMCTorch|  Number of AOs       : 2
INFO:QMCTorch|  Number of MOs       : 2
INFO:QMCTorch|  SCF Energy          : -1.076 Hartree
```

The exhaustive list of supported basis set can be found here : https://pyscf.org/user/gto.html

5.2.2 Slater orbitals with ADF

If a valid SCM license is found QMCTorch can use `ADF`. Two calculators are available depending on the version of ADF installed: * ADF 2019 : `calculator = 'adf2019'` * ADF 2020+ : `calculator = 'adf'`

So for example if ADF2019 is installed the following command will use ADF to compute the electronic structure of the molecule.

[ ]:
```
try:
    mol = Molecule(atom='H 0. 0. 0; H 0. 0. 1.', unit='bohr', calculator='adf2019',␣
↪basis='dzp')
except Exception as expt:
    print(expt)
```

```
INFO:QMCTorch|
INFO:QMCTorch| SCF Calculation
INFO:QMCTorch|  Running scf  calculation
[13.04|16:37:54] PLAMS working folder: /home/nico/QMCTorch/notebooks/plams_workdir.002
File ./plams_workdir/HH_dzp/HH_dzp.t21 not found, ADF may have crashed, look into the␣
↪plams_workdir directory
```

Here as well the `basis` keyword argument specifies the basis set used in the scf calculation. The list of supported basis set can be found here : https://www.scm.com/doc/ADF/Input/Basis_sets_and_atomic_fragments.html

Additional basis sets, namely VB1, VB2, VB3, CVB1, CVB2 and CVB3, are available. These are STO valence and core-valence basis set presented by Ema et. al in "Polarized basis sets for Slater-type orbitals: H to Ne atoms", https://doi.org/10.1002/jcc.10227. Changing the `basis` keyword argument to : ‘‘basis=VB1‘‘‘ will for examle use the small VB1 basis set during the SCF calculation.

5.3 Loading a SCF calculation

By default QMCTorch will create a HDF5 file containing all the required information about the molecule and SCF calculation. The name of this file is given by the name of the molecule, the calculator name and the basis set, e.g. `LiH_adf_dz.hdf5` or `water_pyscf_sto3g.xyz`. This files can be loaded to instanciate the molecule object through the `load` keyword argument:

[12]:
```
mol = Molecule(load='./hdf5/LiH_adf_dz.hdf5')
```

```
INFO:QMCTorch|
INFO:QMCTorch| SCF Calculation
INFO:QMCTorch|  Loading data from LiH_adf_dz.hdf5
```

CHAPTER 6

Sampling and Energy Calculation of a Water Molecule

In this tutorial we explore how to sample the density of a water molecule and compute the total energy of the system using QMC. We first import all the module we will need in the tutorial

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from qmctorch.scf import Molecule
from qmctorch.wavefunction import SlaterJastrow
from qmctorch.sampler import Metropolis
from qmctorch.solver import Solver
from qmctorch.utils import plot_walkers_traj
```

```
INFO:QMCTorch|  _____     __  _________________                _
INFO:QMCTorch| / __ \   /  |/  / ____/_  __/__  _________/ /
INFO:QMCTorch|/ /_/ / / /|_/ / /__   / / / _ \/ __/ __/ _ \
INFO:QMCTorch|\___\_\/_/  /_/\___/ /_/  \___/_/  \__/_//_/
```

6.1 Creating the system

We now create the `Molecule` object. We use here a water molecule and load the coordinates directly from a file. We also need to specify the quantum chemistry package from which we extract the atomic and molecular orbitals information. We choose here `pyscf` and secpify a `sto-3g` basis set.

[2]:
```
# define the molecule
mol = Molecule(atom='water.xyz', unit='angs',
               calculator='pyscf', basis='sto-3g', name='water')
```

```
INFO:QMCTorch|
INFO:QMCTorch| SCF Calculation
INFO:QMCTorch|  Reusing scf results from water_pyscf_sto-3g.hdf5
```

We can now define the wave function ansatz describing the electronic structure of the molecule. We use here a simple `SlaterJastrow` wave function and only include the ground state electronic configuration in the CI expansion. We

use here the default Pade Jastrow form of the Jastrow factor. We therefore have a wave function of the type:

$$\Psi(R) = J(R)D^0_\uparrow(r_\uparrow)D^0_\downarrow(r_\downarrow)$$

[3]:
```
wf = SlaterJastrow(mol, configs='ground_state')
```

```
INFO:QMCTorch|
INFO:QMCTorch| Wave Function
INFO:QMCTorch|  Jastrow factor      : True
INFO:QMCTorch|  Jastrow kernel      : PadeJastrowKernel
INFO:QMCTorch|  Highest MO included : 7
INFO:QMCTorch|  Configurations      : ground_state
INFO:QMCTorch|  Number of confs     : 1
INFO:QMCTorch|  Kinetic energy      : jacobi
INFO:QMCTorch|  Number var  param   : 81
INFO:QMCTorch|  Cuda support        : False
```

We now define the a Metropolis sampler, using only 100 walkers. Each walker contains here the positions of the 10 electrons of molecule. The electrons are initially localized around their atomic center, i.e. 8 around the oxygen atom and 1 around each hydrogen atom. We also specify here that the sampler will perform 500 steps with a step size of 0.25 bohr.

[4]:
```
sampler = Metropolis(nwalkers=100, nstep=500, step_size=0.25,
                     nelec=wf.nelec, ndim=wf.ndim,
                     init=mol.domain('atomic'),
                     move={'type': 'all-elec', 'proba': 'normal'})
```

```
INFO:QMCTorch|
INFO:QMCTorch| Monte-Carlo Sampler
INFO:QMCTorch|  Number of walkers   : 100
INFO:QMCTorch|  Number of steps     : 500
INFO:QMCTorch|  Step size           : 0.25
INFO:QMCTorch|  Thermalization steps: -1
INFO:QMCTorch|  Decorelation steps  : 1
INFO:QMCTorch|  Walkers init pos    : atomic
INFO:QMCTorch|  Move type           : all-elec
INFO:QMCTorch|  Move proba          : normal
```

We can finally initialize the solver that will run the calculation

[5]:
```
solver = Solver(wf=wf, sampler=sampler)
```

```
INFO:QMCTorch|
INFO:QMCTorch| Warning : dump to hdf5
INFO:QMCTorch| Object Solver already exists in water_pyscf_sto-3g_QMCTorch.hdf5
INFO:QMCTorch| Object name changed to SolverSlaterJastrow_5
INFO:QMCTorch|
INFO:QMCTorch|
INFO:QMCTorch| QMC Solver
INFO:QMCTorch|  WaveFunction        : SlaterJastrow
INFO:QMCTorch|  Sampler             : Metropolis
```

6.2 Sampling the density

We can use the wave function and the sampler we just defined to obtain sample of the electronic density. We can for example then plot the positions of the individual electrons in the plane of the molecule to vizually inspect the result of

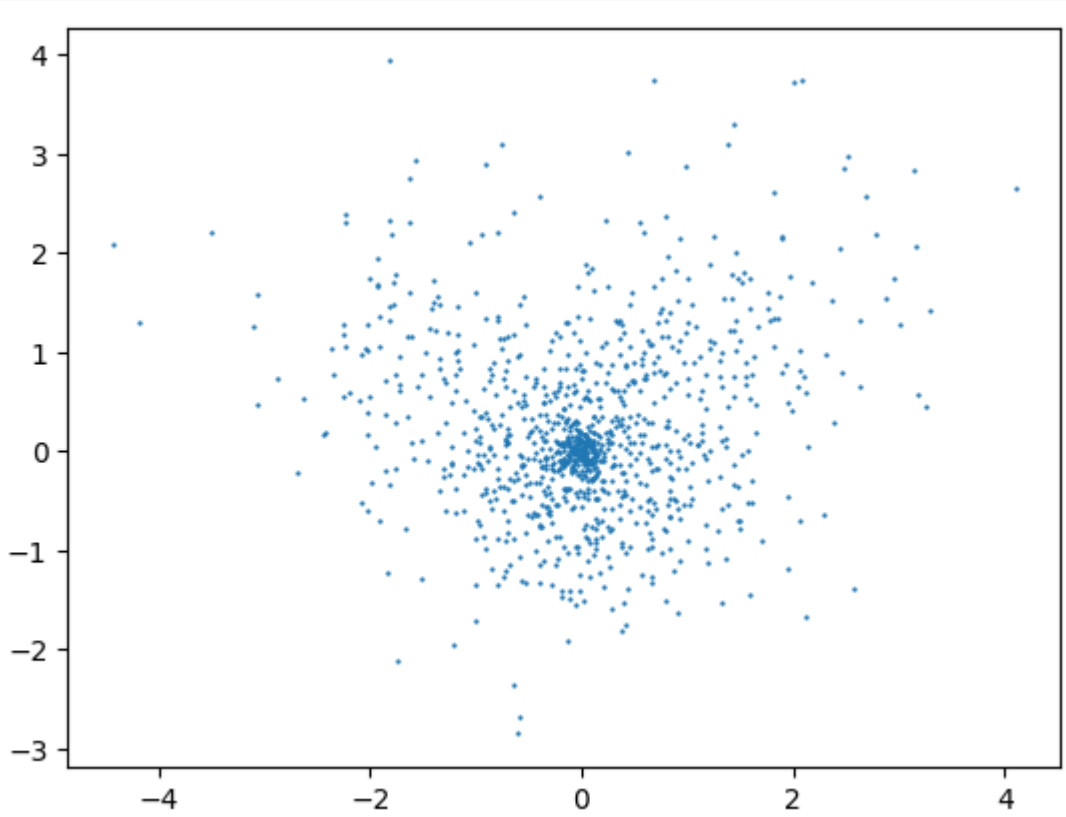
the sampling.

[6]:
```
pos = sampler(wf.pdf)
pos = pos.reshape(100,10,3).cpu().detach().numpy()
plt.scatter(pos[:,:,0],pos[:,:,1],s=0.5)
```

```
INFO:QMCTorch|  Sampling: 100%|| 500/500 [01:46<00:00,  4.70it/s]
```

```
INFO:QMCTorch|   Acceptance rate     : 2.45 %
INFO:QMCTorch|   Timing statistics   : 4.70 steps/sec.
INFO:QMCTorch|   Total Time          : 106.35 sec.
```

[6]:
```
<matplotlib.collections.PathCollection at 0x7fef4e5ea700>
```



6.3 Following indiviudal electron path

By default the sampler only record the position of the electrons at the very last step of the sampling process. We can however change and record all the positions of the electrons during the sampling to be able to track them.

[7]:
```
sampler_singlewalker = Metropolis(nwalkers=1, nstep=500, step_size=0.25,
                      nelec=wf.nelec, ndim=wf.ndim,
                      ntherm=0, ndecor=1,
                      init=mol.domain('atomic'),
                      move={'type': 'all-elec', 'proba': 'normal'})
```

```
INFO:QMCTorch|
INFO:QMCTorch| Monte-Carlo Sampler
```

(continues on next page)

(continued from previous page)

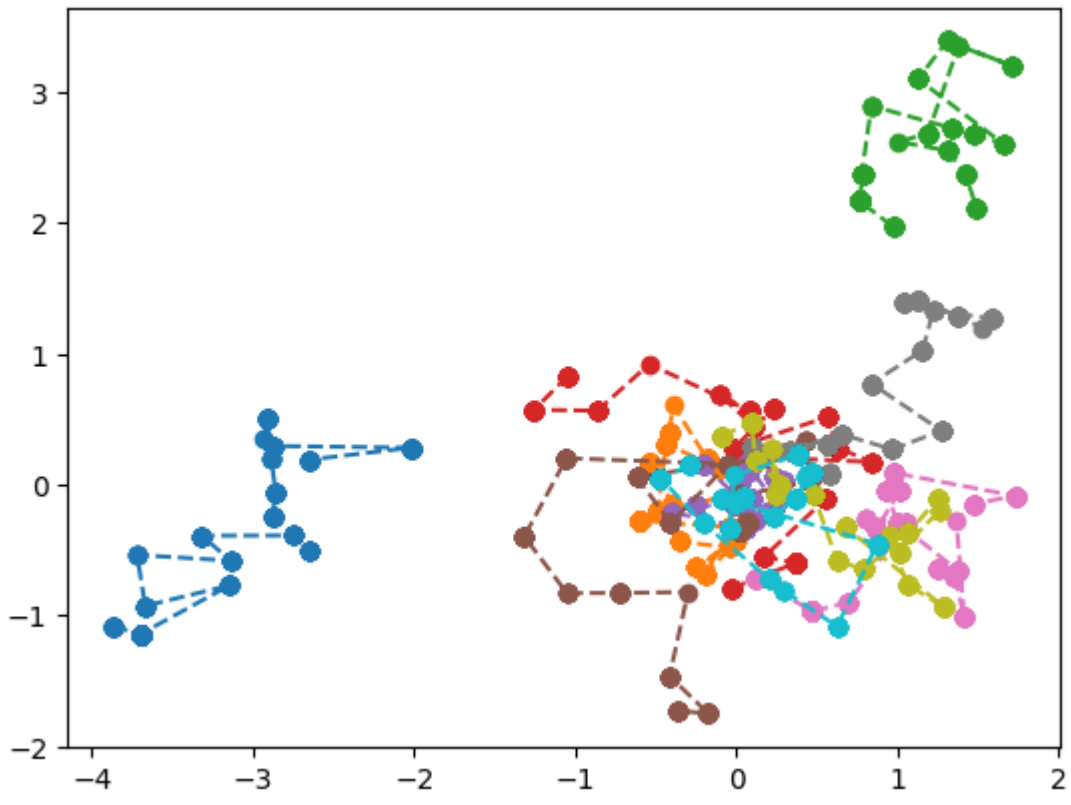
```
INFO:QMCTorch|  Number of walkers   : 1
INFO:QMCTorch|  Number of steps     : 500
INFO:QMCTorch|  Step size           : 0.25
INFO:QMCTorch|  Thermalization steps: 0
INFO:QMCTorch|  Decorelation steps  : 1
INFO:QMCTorch|  Walkers init pos    : atomic
INFO:QMCTorch|  Move type           : all-elec
INFO:QMCTorch|  Move proba          : normal
```

[8]:
```
pos = sampler_singlewalker(wf.pdf)
pos = pos.reshape(-1,10,3).detach().numpy()
plt.plot(pos[:,:,0], pos[:,:,1], marker="o", ls='--')
```

```
INFO:QMCTorch|  Sampling: 100%|| 500/500 [00:20<00:00, 24.27it/s]
```

```
INFO:QMCTorch|   Acceptance rate     : 3.20 %
INFO:QMCTorch|   Timing statistics   : 24.25 steps/sec.
INFO:QMCTorch|   Total Time          : 20.62 sec.
```

[8]:
```
[<matplotlib.lines.Line2D at 0x7fef4cdd4c70>,
 <matplotlib.lines.Line2D at 0x7fef4cdd4d30>,
 <matplotlib.lines.Line2D at 0x7fef4cdd4e20>,
 <matplotlib.lines.Line2D at 0x7fef4cdd4f10>,
 <matplotlib.lines.Line2D at 0x7fef4cde0040>,
 <matplotlib.lines.Line2D at 0x7fef4cde0130>,
 <matplotlib.lines.Line2D at 0x7fef4cde0220>,
 <matplotlib.lines.Line2D at 0x7fef4cde0310>,
 <matplotlib.lines.Line2D at 0x7fef4cde0400>,
 <matplotlib.lines.Line2D at 0x7fef4cde04f0>]
```



6.4 Energy Calculation

To compute the energy of the system we first need to create a solver object that will handle all the orchestration of the calculation

[9]:
```
solver = Solver(wf=wf, sampler=sampler)
```

```
INFO:QMCTorch|
INFO:QMCTorch| Warning : dump to hdf5
INFO:QMCTorch| Object Solver already exists in water_pyscf_sto-3g_QMCTorch.hdf5
INFO:QMCTorch| Object name changed to SolverSlaterJastrow_6
INFO:QMCTorch|
INFO:QMCTorch|
INFO:QMCTorch| QMC Solver
INFO:QMCTorch|  WaveFunction        : SlaterJastrow
INFO:QMCTorch|  Sampler             : Metropolis
```

The energy can then be directly calculated

[10]:
```
obs = solver.single_point()
```

```
INFO:QMCTorch|
INFO:QMCTorch|  Single Point Calculation : 100 walkers | 500 steps
```

```
INFO:QMCTorch|  Sampling: 100%|| 500/500 [01:53<00:00,  4.40it/s]
```

```
INFO:QMCTorch|   Acceptance rate     : 2.36 %
INFO:QMCTorch|   Timing statistics   : 4.40 steps/sec.
INFO:QMCTorch|   Total Time          : 113.76 sec.
INFO:QMCTorch|  Energy   : -72.750839 +/- 1.374327
INFO:QMCTorch|  Variance : 188.877533
INFO:QMCTorch|
INFO:QMCTorch| Warning : dump to hdf5
INFO:QMCTorch| Object single_point already exists in water_pyscf_sto-3g_QMCTorch.hdf5
INFO:QMCTorch| Object name changed to single_point_3
INFO:QMCTorch|
```

6.5 Sampling Trajectory

We can also follow how the total energy thermalize during the sampling process. To this end we need to record the positions of the walkers during the sampling and not just at the end of it. We can then compute the local energies and the total energy at each recorded step of the trajectory. We can either create a new sampler or simply change the configuration of the sampler already included in our solver. We will put the number of thermalization steps to 0 and the number of decorellation step to 5.

[11]:
```
solver.sampler.ntherm = 0
solver.sampler.ndecor = 5
```

We can then resample the density and compute the local energy values along the sampling trajectory and finnaly plot it.

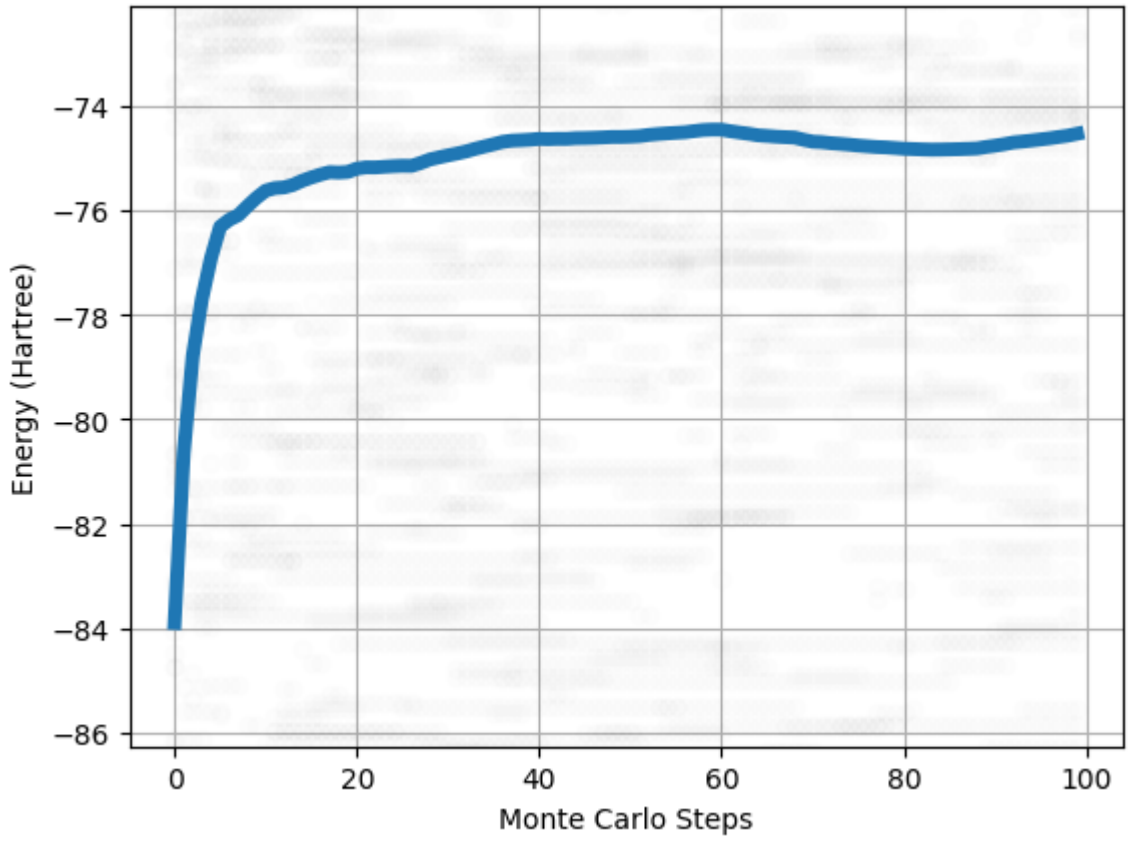
[12]:
```
pos = solver.sampler(solver.wf.pdf)
obs = solver.sampling_traj(pos)
plot_walkers_traj(obs.local_energy, walkers='mean')
```

```
INFO:QMCTorch|  Sampling: 100%|| 500/500 [01:32<00:00,  5.42it/s]
```

```
INFO:QMCTorch|   Acceptance rate      : 2.76 %
INFO:QMCTorch|   Timing statistics    : 5.42 steps/sec.
INFO:QMCTorch|   Total Time           : 92.22 sec.
INFO:QMCTorch|
INFO:QMCTorch|  Sampling trajectory
```

```
INFO:QMCTorch|  Energy  : 100%|| 100/100 [01:06<00:00,  1.50it/s]
```

```
INFO:QMCTorch|
INFO:QMCTorch| Warning : dump to hdf5
INFO:QMCTorch| Object sampling_trajectory already exists in water_pyscf_sto-3g_
↪QMCTorch.hdf5
INFO:QMCTorch| Object name changed to sampling_trajectory_3
INFO:QMCTorch|
```



CHAPTER 7

Wave Function Optimization

We present here a complete example on how to use QMCTorch on a H2 molecule. We first need to import all the relevant modules :

[1]:
```
from torch import optim
from qmctorch.scf import Molecule
from qmctorch.wavefunction import SlaterJastrow
from qmctorch.solver import Solver
from qmctorch.sampler import Metropolis
from qmctorch.utils import set_torch_double_precision
from qmctorch.utils import (plot_energy, plot_data)
set_torch_double_precision()
```

```
INFO:QMCTorch|  _____     __  _________________                _
INFO:QMCTorch| / __ \   /  |/  / ____/_  __/__  _________/ /
INFO:QMCTorch|/ /_/ / / /|_/ / /__   / / / _ \/ __/ __/ _ \
INFO:QMCTorch|\___\_\/_/  /_/\___/ /_/  \___/_/  \__/_//_/
```

7.1 Creating the system

The first step is to define a molecule. We here use a H2 molecule with both hydrgen atoms on the z-axis and separated by 1.38 atomic unit. We choose here to use Slater orbitals that can be otained via `ADF`. We simply here reload calculations to create the molecule

[2]:
```
mol = Molecule(load='./hdf5/H2_adf_dzp.hdf5')
```

```
INFO:QMCTorch|
INFO:QMCTorch| SCF Calculation
INFO:QMCTorch|  Loading data from ./hdf5/H2_adf_dzp.hdf5
```

We then define the wave function relative to this molecule. We also specify here the determinants we want to use in the CI expansion. We use here a to include all the single and double excitation with 2 electrons and 2 orbitals

[3]:
```
wf = SlaterJastrow(mol, configs='single_double(2,2)')
```

```
INFO:QMCTorch|
INFO:QMCTorch| Wave Function
INFO:QMCTorch|  Jastrow factor      : True
INFO:QMCTorch|  Jastrow kernel      : PadeJastrowKernel
INFO:QMCTorch|  Highest MO included : 10
INFO:QMCTorch|  Configurations      : single_double(2,2)
INFO:QMCTorch|  Number of confs     : 4
INFO:QMCTorch|  Kinetic energy      : jacobi
INFO:QMCTorch|  Number var  param   : 121
INFO:QMCTorch|  Cuda support        : False
```

As a sampler we use a simple Metropolis Hasting with 1000 walkers. The walkers are initially localized around the atoms. Each walker will perform 2000 steps of size 0.2 atomic unit and will only keep the last position of each walker (`ntherm=-1`). During each move all the the electrons are moved simultaneously within a normal distribution centered around their current location.

[4]:
```
sampler = Metropolis(nwalkers=5000,
                     nstep=200, step_size=0.2,
                     ntherm=-1, ndecor=100,
                     nelec=wf.nelec, init=mol.domain('atomic'),
                     move={'type': 'all-elec', 'proba': 'normal'})
```

```
INFO:QMCTorch|
INFO:QMCTorch| Monte-Carlo Sampler
INFO:QMCTorch|  Number of walkers   : 5000
INFO:QMCTorch|  Number of steps     : 200
INFO:QMCTorch|  Step size           : 0.2
INFO:QMCTorch|  Thermalization steps: -1
INFO:QMCTorch|  Decorelation steps  : 100
INFO:QMCTorch|  Walkers init pos    : atomic
INFO:QMCTorch|  Move type           : all-elec
INFO:QMCTorch|  Move proba          : normal
```

We will use the ADAM optimizer implemented in pytorch with custom learning rate for each layer. We also define a linear scheduler that will decrease the learning rate after 100 steps

[5]:
```
lr_dict = [{'params': wf.jastrow.parameters(), 'lr': 1E-2},
           {'params': wf.ao.parameters(), 'lr': 1E-6},
           {'params': wf.mo.parameters(), 'lr': 2E-3},
           {'params': wf.fc.parameters(), 'lr': 2E-3}]
opt = optim.Adam(lr_dict, lr=1E-3)
```

A scheduler can also be used to progressively decrease the value of the learning rate during the optimization.

[6]:
```
scheduler = optim.lr_scheduler.StepLR(opt, step_size=100, gamma=0.90)
```

We can now assemble the solver

[7]:
```
solver = Solver(wf=wf, sampler=sampler, optimizer=opt, scheduler=None)
```

```
INFO:QMCTorch|
INFO:QMCTorch| Warning : dump to hdf5
INFO:QMCTorch| Object Solver already exists in H2_adf_dzp_QMCTorch.hdf5
INFO:QMCTorch| Object name changed to SolverSlaterJastrow_7
INFO:QMCTorch|
```

(continues on next page)

(continued from previous page)

```
INFO:QMCTorch|
INFO:QMCTorch| QMC Solver
INFO:QMCTorch|  WaveFunction        : SlaterJastrow
INFO:QMCTorch|  Sampler             : Metropolis
INFO:QMCTorch|  Optimizer           : Adam
```

7.2 Comfiguring the solver

Many parameters of the optimization can be controlled. We can specify which observale to track during the optimization. Here only the local energies will be recorded but one can also record the variational parameters

[8]:
```
solver.configure(track=['local_energy', 'parameters'])
```

Some variational parameters can be frozen and therefore not optimized. We here freeze the MO coefficients and the AO parameters and therefore only the jastrow parametres and the CI coefficients will be optmized

[9]:
```
solver.configure(freeze=['ao'])
```

Either the mean or the variance of local energies can be used as a loss function. We choose here to minimize the energy to optimize the wave function

[10]:
```
solver.configure(loss='energy')
```

The gradients of the wave function w.r.t. the variational parameters can be computed directly via automatic differntiation (`grad='auto'`)or manually (`grad='auto'`) via a reduced noise formula. We pick here a manual calculation

[11]:
```
solver.configure(grad='manual')
```

We also configure the resampling so that the positions of the walkers are updated by performing 25 MC steps from their previous positions after each optimization step.

[12]:
```
solver.configure(resampling={'mode': 'update',
                             'resample_every': 1,
                             'nstep_update': 25})
```

7.3 Running the wave function optimization

We can now run the optimization. We use here 250 optimization steps (epoch), using all the points in a single mini-batch.

[13]:
```
obs = solver.run(50)
```

```
INFO:QMCTorch|
INFO:QMCTorch|  Optimization
INFO:QMCTorch|  Task                :
INFO:QMCTorch|  Number Parameters   : 115
INFO:QMCTorch|  Number of epoch     : 50
INFO:QMCTorch|  Batch size          : 5000
INFO:QMCTorch|  Loss function       : energy
INFO:QMCTorch|  Clip Loss           : False
INFO:QMCTorch|  Gradients           : manual
```

(continues on next page)

(continued from previous page)

```
INFO:QMCTorch|  Resampling mode     : update
INFO:QMCTorch|  Resampling every    : 1
INFO:QMCTorch|  Resampling steps    : 25
INFO:QMCTorch|  Output file         : H2_adf_dzp_QMCTorch.hdf5
INFO:QMCTorch|  Checkpoint every    : None
INFO:QMCTorch|
INFO:QMCTorch|
INFO:QMCTorch|  epoch 0
INFO:QMCTorch|  energy   : -1.155363 +/- 0.003267
INFO:QMCTorch|  variance : 0.231010
INFO:QMCTorch|  epoch done in 0.49 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 1
INFO:QMCTorch|  energy   : -1.149161 +/- 0.003279
INFO:QMCTorch|  variance : 0.231844
INFO:QMCTorch|  epoch done in 0.59 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 2
INFO:QMCTorch|  energy   : -1.150710 +/- 0.003106
INFO:QMCTorch|  variance : 0.219625
INFO:QMCTorch|  epoch done in 0.94 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 3
INFO:QMCTorch|  energy   : -1.156548 +/- 0.003170
INFO:QMCTorch|  variance : 0.224149
INFO:QMCTorch|  epoch done in 0.50 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 4
INFO:QMCTorch|  energy   : -1.155115 +/- 0.003221
INFO:QMCTorch|  variance : 0.227777
INFO:QMCTorch|  epoch done in 0.51 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 5
INFO:QMCTorch|  energy   : -1.156112 +/- 0.003083
INFO:QMCTorch|  variance : 0.217972
INFO:QMCTorch|  epoch done in 0.51 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 6
INFO:QMCTorch|  energy   : -1.155542 +/- 0.003070
INFO:QMCTorch|  variance : 0.217062
INFO:QMCTorch|  epoch done in 0.94 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 7
INFO:QMCTorch|  energy   : -1.157297 +/- 0.003046
INFO:QMCTorch|  variance : 0.215387
INFO:QMCTorch|  epoch done in 0.48 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 8
INFO:QMCTorch|  energy   : -1.150183 +/- 0.003147
INFO:QMCTorch|  variance : 0.222538
INFO:QMCTorch|  epoch done in 0.50 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 9
INFO:QMCTorch|  energy   : -1.155700 +/- 0.003062
INFO:QMCTorch|  variance : 0.216530
INFO:QMCTorch|  epoch done in 0.50 sec.
INFO:QMCTorch|
```

(continues on next page)

(continued from previous page)

```
INFO:QMCTorch|  epoch 10
INFO:QMCTorch|  energy   : -1.154875 +/- 0.003005
INFO:QMCTorch|  variance : 0.212476
INFO:QMCTorch|  epoch done in 0.60 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 11
INFO:QMCTorch|  energy   : -1.154984 +/- 0.003024
INFO:QMCTorch|  variance : 0.213820
INFO:QMCTorch|  epoch done in 0.50 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 12
INFO:QMCTorch|  energy   : -1.154497 +/- 0.002974
INFO:QMCTorch|  variance : 0.210262
INFO:QMCTorch|  epoch done in 0.50 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 13
INFO:QMCTorch|  energy   : -1.157227 +/- 0.003000
INFO:QMCTorch|  variance : 0.212123
INFO:QMCTorch|  epoch done in 0.57 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 14
INFO:QMCTorch|  energy   : -1.156778 +/- 0.002914
INFO:QMCTorch|  variance : 0.206054
INFO:QMCTorch|  epoch done in 0.75 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 15
INFO:QMCTorch|  energy   : -1.152052 +/- 0.003022
INFO:QMCTorch|  variance : 0.213717
INFO:QMCTorch|  epoch done in 0.49 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 16
INFO:QMCTorch|  energy   : -1.158149 +/- 0.002847
INFO:QMCTorch|  variance : 0.201333
INFO:QMCTorch|  epoch done in 0.50 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 17
INFO:QMCTorch|  energy   : -1.158337 +/- 0.002852
INFO:QMCTorch|  variance : 0.201654
INFO:QMCTorch|  epoch done in 0.48 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 18
INFO:QMCTorch|  energy   : -1.158138 +/- 0.002793
INFO:QMCTorch|  variance : 0.197500
INFO:QMCTorch|  epoch done in 0.89 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 19
INFO:QMCTorch|  energy   : -1.157327 +/- 0.002869
INFO:QMCTorch|  variance : 0.202897
INFO:QMCTorch|  epoch done in 0.99 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 20
INFO:QMCTorch|  energy   : -1.155671 +/- 0.002901
INFO:QMCTorch|  variance : 0.205139
INFO:QMCTorch|  epoch done in 0.52 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 21
INFO:QMCTorch|  energy   : -1.156606 +/- 0.002863
```

(continues on next page)

(continued from previous page)

```
INFO:QMCTorch|  variance : 0.202470
INFO:QMCTorch|  epoch done in 0.48 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 22
INFO:QMCTorch|  energy   : -1.164993 +/- 0.002852
INFO:QMCTorch|  variance : 0.201661
INFO:QMCTorch|  epoch done in 0.51 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 23
INFO:QMCTorch|  energy   : -1.157040 +/- 0.002765
INFO:QMCTorch|  variance : 0.195510
INFO:QMCTorch|  epoch done in 0.50 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 24
INFO:QMCTorch|  energy   : -1.163667 +/- 0.002707
INFO:QMCTorch|  variance : 0.191386
INFO:QMCTorch|  epoch done in 0.57 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 25
INFO:QMCTorch|  energy   : -1.159113 +/- 0.002700
INFO:QMCTorch|  variance : 0.190943
INFO:QMCTorch|  epoch done in 0.51 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 26
INFO:QMCTorch|  energy   : -1.162071 +/- 0.002661
INFO:QMCTorch|  variance : 0.188190
INFO:QMCTorch|  epoch done in 0.53 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 27
INFO:QMCTorch|  energy   : -1.158837 +/- 0.002642
INFO:QMCTorch|  variance : 0.186836
INFO:QMCTorch|  epoch done in 0.49 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 28
INFO:QMCTorch|  energy   : -1.155956 +/- 0.002649
INFO:QMCTorch|  variance : 0.187284
INFO:QMCTorch|  epoch done in 0.50 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 29
INFO:QMCTorch|  energy   : -1.162127 +/- 0.002609
INFO:QMCTorch|  variance : 0.184491
INFO:QMCTorch|  epoch done in 0.73 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 30
INFO:QMCTorch|  energy   : -1.163752 +/- 0.002560
INFO:QMCTorch|  variance : 0.181025
INFO:QMCTorch|  epoch done in 0.52 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 31
INFO:QMCTorch|  energy   : -1.159163 +/- 0.002590
INFO:QMCTorch|  variance : 0.183165
INFO:QMCTorch|  epoch done in 0.56 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 32
INFO:QMCTorch|  energy   : -1.163472 +/- 0.002603
INFO:QMCTorch|  variance : 0.184072
INFO:QMCTorch|  epoch done in 0.50 sec.
```

(continues on next page)

(continued from previous page)

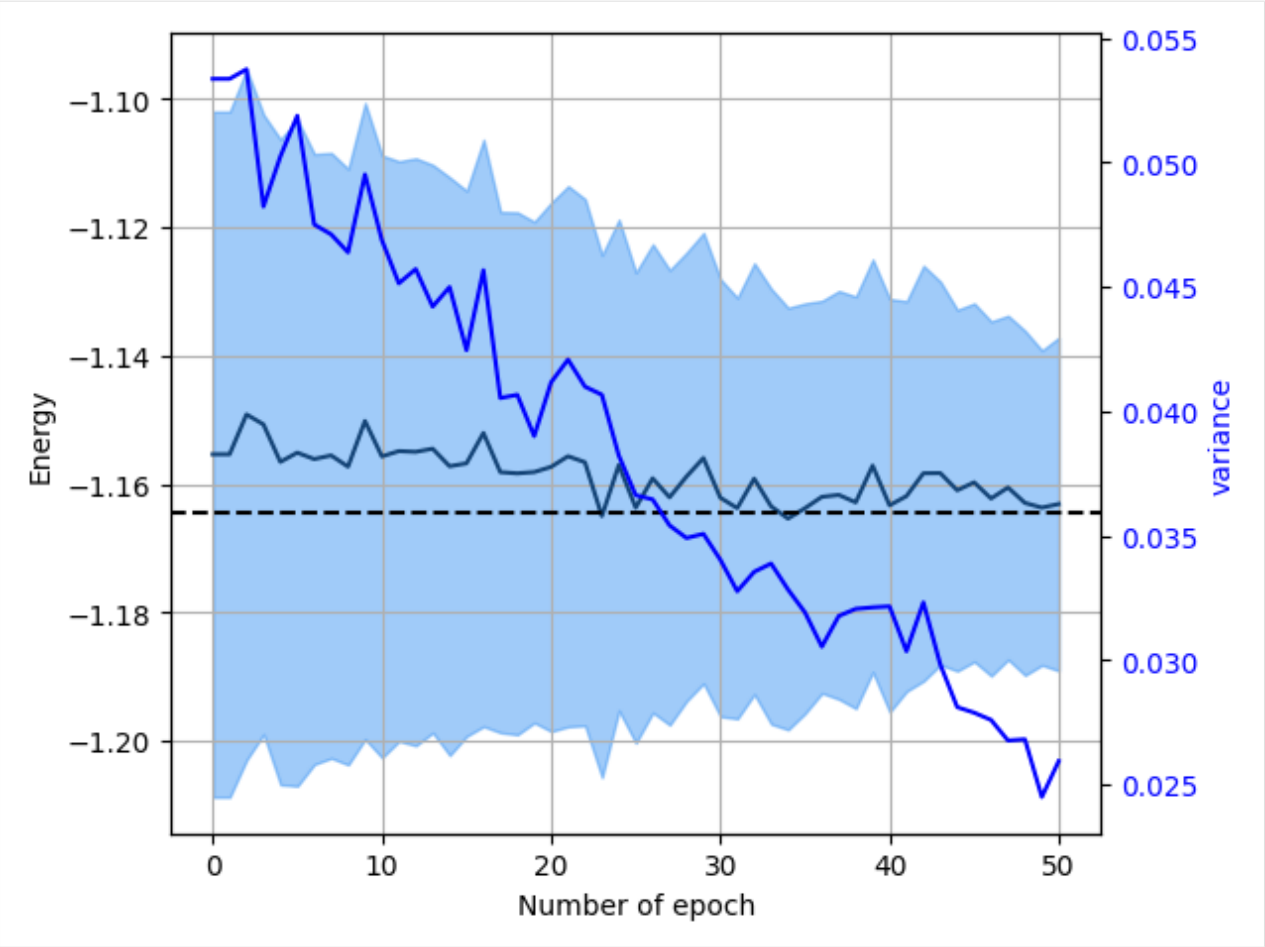
```
INFO:QMCTorch|
INFO:QMCTorch|  epoch 33
INFO:QMCTorch|  energy   : -1.165384 +/- 0.002563
INFO:QMCTorch|  variance : 0.181214
INFO:QMCTorch|  epoch done in 0.51 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 34
INFO:QMCTorch|  energy   : -1.163774 +/- 0.002527
INFO:QMCTorch|  variance : 0.178661
INFO:QMCTorch|  epoch done in 0.50 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 35
INFO:QMCTorch|  energy   : -1.161995 +/- 0.002472
INFO:QMCTorch|  variance : 0.174763
INFO:QMCTorch|  epoch done in 0.51 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 36
INFO:QMCTorch|  energy   : -1.161698 +/- 0.002521
INFO:QMCTorch|  variance : 0.178254
INFO:QMCTorch|  epoch done in 0.50 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 37
INFO:QMCTorch|  energy   : -1.162856 +/- 0.002532
INFO:QMCTorch|  variance : 0.179051
INFO:QMCTorch|  epoch done in 0.50 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 38
INFO:QMCTorch|  energy   : -1.157138 +/- 0.002535
INFO:QMCTorch|  variance : 0.179220
INFO:QMCTorch|  epoch done in 0.50 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 39
INFO:QMCTorch|  energy   : -1.163320 +/- 0.002536
INFO:QMCTorch|  variance : 0.179332
INFO:QMCTorch|  epoch done in 0.74 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 40
INFO:QMCTorch|  energy   : -1.161880 +/- 0.002464
INFO:QMCTorch|  variance : 0.174239
INFO:QMCTorch|  epoch done in 0.48 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 41
INFO:QMCTorch|  energy   : -1.158324 +/- 0.002542
INFO:QMCTorch|  variance : 0.179777
INFO:QMCTorch|  epoch done in 0.51 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 42
INFO:QMCTorch|  energy   : -1.158298 +/- 0.002442
INFO:QMCTorch|  variance : 0.172696
INFO:QMCTorch|  epoch done in 0.50 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 43
INFO:QMCTorch|  energy   : -1.160970 +/- 0.002371
INFO:QMCTorch|  variance : 0.167662
INFO:QMCTorch|  epoch done in 0.79 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 44
```

(continues on next page)

(continued from previous page)

```
INFO:QMCTorch|  energy   : -1.159741 +/- 0.002362
INFO:QMCTorch|  variance : 0.166993
INFO:QMCTorch|  epoch done in 0.51 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 45
INFO:QMCTorch|  energy   : -1.162254 +/- 0.002349
INFO:QMCTorch|  variance : 0.166119
INFO:QMCTorch|  epoch done in 0.73 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 46
INFO:QMCTorch|  energy   : -1.160540 +/- 0.002314
INFO:QMCTorch|  variance : 0.163611
INFO:QMCTorch|  epoch done in 0.49 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 47
INFO:QMCTorch|  energy   : -1.162938 +/- 0.002316
INFO:QMCTorch|  variance : 0.163749
INFO:QMCTorch|  epoch done in 0.49 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 48
INFO:QMCTorch|  energy   : -1.163674 +/- 0.002214
INFO:QMCTorch|  variance : 0.156522
INFO:QMCTorch|  epoch done in 0.51 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 49
INFO:QMCTorch|  energy   : -1.163112 +/- 0.002278
INFO:QMCTorch|  variance : 0.161065
INFO:QMCTorch|  epoch done in 0.50 sec.
INFO:QMCTorch|
INFO:QMCTorch| Warning : dump to hdf5
INFO:QMCTorch| Object wf_opt already exists in H2_adf_dzp_QMCTorch.hdf5
INFO:QMCTorch| Object name changed to wf_opt_7
INFO:QMCTorch|
```

[14]:
```
plot_energy(obs.local_energy, e0=-1.1645, show_variance=True)
```


−1.10
−1.12
−1.14
−1.16
−1.18
−1.20
Energy
0.055
0.050
0.045
0.040
0.035
0.030
0.025
variance
0
10
20
30
40
50
Number of epoch

[ ]:

CHAPTER 8

Geometry Optimization

We present here a complete example on how to use QMCTorch to optimize the geometry of a H2 molecule. As previously the firs task is to import all the modules needed

[1]:
```
from torch import optim
from torch.optim import Adam
from qmctorch.wavefunction import SlaterJastrow
from qmctorch.solver import Solver
from qmctorch.sampler import Metropolis
from qmctorch.scf import Molecule
from qmctorch.utils import plot_energy
from qmctorch.utils import set_torch_double_precision
set_torch_double_precision()
```

```
INFO:QMCTorch|  _____     __  _________________                _
INFO:QMCTorch| / __ \   /  |/  / ____/_  __/__  _________/ /
INFO:QMCTorch|/ /_/ / / /|_/ / /__   / / / _ \/ __/ __/ _ \
INFO:QMCTorch|\___\_\/_/  /_/\___/ /_/  \___/_/  \__/_//_/
```

We then define the molecule. We create a H2 molecule with a short atomic distance that we will then try to relax

[2]:
```
mol = Molecule(atom = 'H 0. 0. -0.5; H 0. 0. 0.5', unit='bohr',
               calculator='pyscf', basis='sto-3g', redo_scf=True)
```

```
INFO:QMCTorch|
INFO:QMCTorch| SCF Calculation
INFO:QMCTorch|  Removing H2_pyscf_sto-3g.hdf5 and redo SCF calculations
INFO:QMCTorch|  Running scf  calculation
converged SCF energy = -1.06599946214331
INFO:QMCTorch|  Molecule name       : H2
INFO:QMCTorch|  Number of electrons : 2
INFO:QMCTorch|  SCF calculator      : pyscf
INFO:QMCTorch|  Basis set           : sto-3g
INFO:QMCTorch|  SCF                 : HF
INFO:QMCTorch|  Number of AOs       : 2
INFO:QMCTorch|  Number of MOs       : 2
INFO:QMCTorch|  SCF Energy          : -1.066 Hartree
```

As for the wave function optimization example we need to define a wave function, a sampler an optimizer and a solver.

[3]:
```
# wave function with only the ground state determinant
wf = SlaterJastrow(mol, configs='single_double(2,2)').gto2sto()

# sampler
sampler = Metropolis(nwalkers=1000, nstep=200, step_size=0.5,
                     nelec=wf.nelec, ndim=wf.ndim,
                     ntherm=-1, ndecor=100,
                     init=mol.domain('normal'),
                     move={'type': 'all-elec', 'proba': 'normal'})
# optimizer
opt = Adam(wf.parameters(), lr=0.005)

# scheduler
scheduler = optim.lr_scheduler.StepLR(opt, step_size=20, gamma=0.75)
```

```
INFO:QMCTorch|
INFO:QMCTorch| Wave Function
INFO:QMCTorch|  Jastrow factor      : True
INFO:QMCTorch|  Jastrow kernel      : PadeJastrowKernel
INFO:QMCTorch|  Highest MO included : 2
INFO:QMCTorch|  Configurations      : single_double(2,2)
INFO:QMCTorch|  Number of confs     : 4
INFO:QMCTorch|  Kinetic energy      : jacobi
INFO:QMCTorch|  Number var  param   : 21
INFO:QMCTorch|  Cuda support        : False
INFO:QMCTorch|  Fit GTOs to STOs  :
INFO:QMCTorch|
INFO:QMCTorch| Wave Function
INFO:QMCTorch|  Jastrow factor      : True
INFO:QMCTorch|  Jastrow kernel      : PadeJastrowKernel
INFO:QMCTorch|  Highest MO included : 2
INFO:QMCTorch|  Configurations      : single_double(2,2)
INFO:QMCTorch|  Number of confs     : 4
INFO:QMCTorch|  Kinetic energy      : jacobi
INFO:QMCTorch|  Number var  param   : 17
INFO:QMCTorch|  Cuda support        : False
INFO:QMCTorch|
INFO:QMCTorch| Monte-Carlo Sampler
INFO:QMCTorch|  Number of walkers   : 1000
INFO:QMCTorch|  Number of steps     : 200
INFO:QMCTorch|  Step size           : 0.5
INFO:QMCTorch|  Thermalization steps: -1
INFO:QMCTorch|  Decorelation steps  : 100
INFO:QMCTorch|  Walkers init pos    : normal
INFO:QMCTorch|  Move type           : all-elec
INFO:QMCTorch|  Move proba          : normal
```

[4]:
```
solver = Solver(wf=wf,
                      sampler=sampler,
                      optimizer=opt,
                      scheduler=None)
solver.configure(loss='energy', grad='auto', track=['local_energy','geometry'],␣
↪freeze = ['ao'])
```

```
INFO:QMCTorch|
INFO:QMCTorch| Warning : dump to hdf5
INFO:QMCTorch| Object Solver already exists in H2_pyscf_sto-3g_QMCTorch.hdf5
```

(continues on next page)

(continued from previous page)

```
INFO:QMCTorch| Object name changed to SolverSlaterJastrow_9
INFO:QMCTorch|
INFO:QMCTorch|
INFO:QMCTorch| QMC Solver
INFO:QMCTorch|  WaveFunction        : SlaterJastrow
INFO:QMCTorch|  Sampler             : Metropolis
INFO:QMCTorch|  Optimizer           : Adam
```

Note that we must set `grad='auto'` to perform a geometry optimization. To optimize the geometry of the molecule we must use the `set_params_requires_grad` method of the solver. We here set all the wave function parameters and all the atomic positions as variational parameters, leading to a simultaneous optimization of the wave function and of the atomic coordinates. We can then run the optimization here using 50 epochs

[5]:
```
solver.set_params_requires_grad(wf_params=False, geo_params=True)
obs = solver.run(50)
```

```
INFO:QMCTorch|
INFO:QMCTorch|  Optimization
INFO:QMCTorch|  Task                :
INFO:QMCTorch|  Number Parameters   : 6
INFO:QMCTorch|  Number of epoch     : 50
INFO:QMCTorch|  Batch size          : 1000
INFO:QMCTorch|  Loss function       : energy
INFO:QMCTorch|  Clip Loss           : False
INFO:QMCTorch|  Gradients           : auto
INFO:QMCTorch|  Resampling mode     : update
INFO:QMCTorch|  Resampling every    : 1
INFO:QMCTorch|  Resampling steps    : 25
INFO:QMCTorch|  Output file         : H2_pyscf_sto-3g_QMCTorch.hdf5
INFO:QMCTorch|  Checkpoint every    : None
INFO:QMCTorch|
INFO:QMCTorch|
INFO:QMCTorch|  epoch 0
INFO:QMCTorch|  energy   : -1.070846 +/- 0.012322
INFO:QMCTorch|  variance : 0.389654
INFO:QMCTorch|  epoch done in 0.09 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 1
INFO:QMCTorch|  energy   : -1.093524 +/- 0.012193
INFO:QMCTorch|  variance : 0.385562
INFO:QMCTorch|  epoch done in 0.09 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 2
INFO:QMCTorch|  energy   : -1.085855 +/- 0.011676
INFO:QMCTorch|  variance : 0.369217
INFO:QMCTorch|  epoch done in 0.12 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 3
INFO:QMCTorch|  energy   : -1.095264 +/- 0.011571
INFO:QMCTorch|  variance : 0.365905
INFO:QMCTorch|  epoch done in 0.10 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 4
INFO:QMCTorch|  energy   : -1.078411 +/- 0.010946
INFO:QMCTorch|  variance : 0.346155
INFO:QMCTorch|  epoch done in 0.34 sec.
```

(continues on next page)

(continued from previous page)

```
INFO:QMCTorch|
INFO:QMCTorch|  epoch 5
INFO:QMCTorch|  energy   : -1.073701 +/- 0.011750
INFO:QMCTorch|  variance : 0.371554
INFO:QMCTorch|  epoch done in 0.35 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 6
INFO:QMCTorch|  energy   : -1.083239 +/- 0.011441
INFO:QMCTorch|  variance : 0.361785
INFO:QMCTorch|  epoch done in 0.10 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 7
INFO:QMCTorch|  energy   : -1.091864 +/- 0.011191
INFO:QMCTorch|  variance : 0.353902
INFO:QMCTorch|  epoch done in 0.13 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 8
INFO:QMCTorch|  energy   : -1.112321 +/- 0.010893
INFO:QMCTorch|  variance : 0.344459
INFO:QMCTorch|  epoch done in 0.10 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 9
INFO:QMCTorch|  energy   : -1.101314 +/- 0.011203
INFO:QMCTorch|  variance : 0.354284
INFO:QMCTorch|  epoch done in 0.09 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 10
INFO:QMCTorch|  energy   : -1.095483 +/- 0.011385
INFO:QMCTorch|  variance : 0.360013
INFO:QMCTorch|  epoch done in 0.12 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 11
INFO:QMCTorch|  energy   : -1.120974 +/- 0.010508
INFO:QMCTorch|  variance : 0.332293
INFO:QMCTorch|  epoch done in 0.10 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 12
INFO:QMCTorch|  energy   : -1.116147 +/- 0.010295
INFO:QMCTorch|  variance : 0.325555
INFO:QMCTorch|  epoch done in 0.12 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 13
INFO:QMCTorch|  energy   : -1.090706 +/- 0.011097
INFO:QMCTorch|  variance : 0.350910
INFO:QMCTorch|  epoch done in 0.10 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 14
INFO:QMCTorch|  energy   : -1.106975 +/- 0.010232
INFO:QMCTorch|  variance : 0.323554
INFO:QMCTorch|  epoch done in 0.09 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 15
INFO:QMCTorch|  energy   : -1.114616 +/- 0.010598
INFO:QMCTorch|  variance : 0.335125
INFO:QMCTorch|  epoch done in 0.11 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 16
```

(continues on next page)

(continued from previous page)

```
INFO:QMCTorch|  energy   : -1.120063 +/- 0.009871
INFO:QMCTorch|  variance : 0.312143
INFO:QMCTorch|  epoch done in 0.10 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 17
INFO:QMCTorch|  energy   : -1.126814 +/- 0.010350
INFO:QMCTorch|  variance : 0.327289
INFO:QMCTorch|  epoch done in 0.11 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 18
INFO:QMCTorch|  energy   : -1.110957 +/- 0.009913
INFO:QMCTorch|  variance : 0.313466
INFO:QMCTorch|  epoch done in 0.10 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 19
INFO:QMCTorch|  energy   : -1.135297 +/- 0.009999
INFO:QMCTorch|  variance : 0.316206
INFO:QMCTorch|  epoch done in 0.10 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 20
INFO:QMCTorch|  energy   : -1.126248 +/- 0.010025
INFO:QMCTorch|  variance : 0.317012
INFO:QMCTorch|  epoch done in 0.13 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 21
INFO:QMCTorch|  energy   : -1.119559 +/- 0.010067
INFO:QMCTorch|  variance : 0.318341
INFO:QMCTorch|  epoch done in 0.10 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 22
INFO:QMCTorch|  energy   : -1.114980 +/- 0.009557
INFO:QMCTorch|  variance : 0.302212
INFO:QMCTorch|  epoch done in 0.11 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 23
INFO:QMCTorch|  energy   : -1.122430 +/- 0.010054
INFO:QMCTorch|  variance : 0.317943
INFO:QMCTorch|  epoch done in 0.09 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 24
INFO:QMCTorch|  energy   : -1.122087 +/- 0.009651
INFO:QMCTorch|  variance : 0.305191
INFO:QMCTorch|  epoch done in 0.11 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 25
INFO:QMCTorch|  energy   : -1.133653 +/- 0.009697
INFO:QMCTorch|  variance : 0.306655
INFO:QMCTorch|  epoch done in 0.10 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 26
INFO:QMCTorch|  energy   : -1.153086 +/- 0.009464
INFO:QMCTorch|  variance : 0.299277
INFO:QMCTorch|  epoch done in 0.09 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 27
INFO:QMCTorch|  energy   : -1.137196 +/- 0.009235
INFO:QMCTorch|  variance : 0.292042
```

(continues on next page)

(continued from previous page)

```
INFO:QMCTorch|  epoch done in 0.11 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 28
INFO:QMCTorch|  energy   : -1.134850 +/- 0.009741
INFO:QMCTorch|  variance : 0.308045
INFO:QMCTorch|  epoch done in 0.09 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 29
INFO:QMCTorch|  energy   : -1.137292 +/- 0.008938
INFO:QMCTorch|  variance : 0.282648
INFO:QMCTorch|  epoch done in 0.11 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 30
INFO:QMCTorch|  energy   : -1.129972 +/- 0.009542
INFO:QMCTorch|  variance : 0.301736
INFO:QMCTorch|  epoch done in 0.09 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 31
INFO:QMCTorch|  energy   : -1.129879 +/- 0.008805
INFO:QMCTorch|  variance : 0.278450
INFO:QMCTorch|  epoch done in 0.09 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 32
INFO:QMCTorch|  energy   : -1.120897 +/- 0.009573
INFO:QMCTorch|  variance : 0.302713
INFO:QMCTorch|  epoch done in 0.11 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 33
INFO:QMCTorch|  energy   : -1.126543 +/- 0.008883
INFO:QMCTorch|  variance : 0.280912
INFO:QMCTorch|  epoch done in 0.09 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 34
INFO:QMCTorch|  energy   : -1.125339 +/- 0.009391
INFO:QMCTorch|  variance : 0.296973
INFO:QMCTorch|  epoch done in 0.11 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 35
INFO:QMCTorch|  energy   : -1.132248 +/- 0.009047
INFO:QMCTorch|  variance : 0.286092
INFO:QMCTorch|  epoch done in 0.11 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 36
INFO:QMCTorch|  energy   : -1.143987 +/- 0.008693
INFO:QMCTorch|  variance : 0.274895
INFO:QMCTorch|  epoch done in 0.09 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 37
INFO:QMCTorch|  energy   : -1.143939 +/- 0.009039
INFO:QMCTorch|  variance : 0.285829
INFO:QMCTorch|  epoch done in 0.11 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 38
INFO:QMCTorch|  energy   : -1.132018 +/- 0.008930
INFO:QMCTorch|  variance : 0.282378
INFO:QMCTorch|  epoch done in 0.10 sec.
INFO:QMCTorch|
```

(continues on next page)

(continued from previous page)

```
INFO:QMCTorch|  epoch 39
INFO:QMCTorch|  energy   : -1.135228 +/- 0.008902
INFO:QMCTorch|  variance : 0.281495
INFO:QMCTorch|  epoch done in 0.11 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 40
INFO:QMCTorch|  energy   : -1.146552 +/- 0.009092
INFO:QMCTorch|  variance : 0.287503
INFO:QMCTorch|  epoch done in 0.10 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 41
INFO:QMCTorch|  energy   : -1.141178 +/- 0.008888
INFO:QMCTorch|  variance : 0.281070
INFO:QMCTorch|  epoch done in 0.10 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 42
INFO:QMCTorch|  energy   : -1.133022 +/- 0.009063
INFO:QMCTorch|  variance : 0.286591
INFO:QMCTorch|  epoch done in 0.11 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 43
INFO:QMCTorch|  energy   : -1.133298 +/- 0.008375
INFO:QMCTorch|  variance : 0.264828
INFO:QMCTorch|  epoch done in 0.23 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 44
INFO:QMCTorch|  energy   : -1.135157 +/- 0.008890
INFO:QMCTorch|  variance : 0.281116
INFO:QMCTorch|  epoch done in 0.38 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 45
INFO:QMCTorch|  energy   : -1.143128 +/- 0.008901
INFO:QMCTorch|  variance : 0.281471
INFO:QMCTorch|  epoch done in 0.09 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 46
INFO:QMCTorch|  energy   : -1.148810 +/- 0.008243
INFO:QMCTorch|  variance : 0.260676
INFO:QMCTorch|  epoch done in 0.12 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 47
INFO:QMCTorch|  energy   : -1.143023 +/- 0.008719
INFO:QMCTorch|  variance : 0.275712
INFO:QMCTorch|  epoch done in 0.28 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 48
INFO:QMCTorch|  energy   : -1.154847 +/- 0.008289
INFO:QMCTorch|  variance : 0.262111
INFO:QMCTorch|  epoch done in 0.33 sec.
INFO:QMCTorch|
INFO:QMCTorch|  epoch 49
INFO:QMCTorch|  energy   : -1.140911 +/- 0.008209
INFO:QMCTorch|  variance : 0.259583
INFO:QMCTorch|  epoch done in 0.24 sec.
INFO:QMCTorch|
INFO:QMCTorch| Warning : dump to hdf5
INFO:QMCTorch| Object wf_opt already exists in H2_pyscf_sto-3g_QMCTorch.hdf5
```

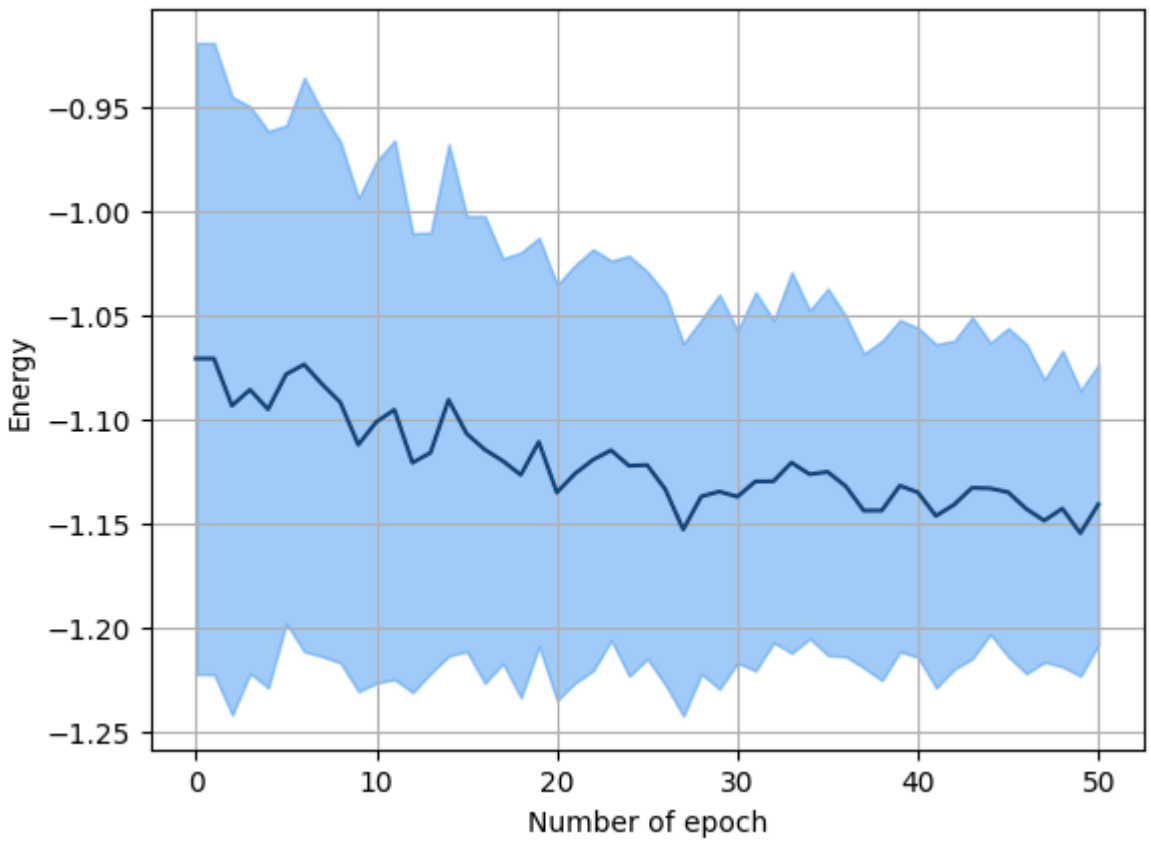
(continues on next page)

(continued from previous page)

```
INFO:QMCTorch| Object name changed to wf_opt_5
INFO:QMCTorch|
```

[6]:
```
solver.save_traj('h2_traj.xyz', obs)
```

[8]:
```
plot_energy(obs.local_energy)
```



[ ]:

CHAPTER 9

Single GPU Support

Warning The use of GPU and mutli-GPU is under developpement and hasn't been thoroughly tested yet. Proceed with caution !

Using pytorch as a backend, QMCTorch can leverage GPU cards available on your hardware. You of course must have the CUDA version of pytorch installed (https://pytorch.org/)

Let's first import everything and create a molecule

[13]:
```
import torch
from torch import optim
from qmctorch.scf import Molecule
from qmctorch.wavefunction import SlaterJastrow
from qmctorch.sampler import Metropolis
from qmctorch.utils import (plot_energy, plot_data)
mol = Molecule(atom='H 0. 0. 0; H 0. 0. 1.', unit='bohr', redo_scf=True)
```

```
INFO:QMCTorch|
INFO:QMCTorch| SCF Calculation
INFO:QMCTorch|  Removing H2_pyscf_sto-3g.hdf5 and redo SCF calculations
INFO:QMCTorch|  Running scf  calculation
converged SCF energy = -1.06599946214331
INFO:QMCTorch|  Molecule name       : H2
INFO:QMCTorch|  Number of electrons : 2
INFO:QMCTorch|  SCF calculator      : pyscf
INFO:QMCTorch|  Basis set           : sto-3g
INFO:QMCTorch|  SCF                 : HF
INFO:QMCTorch|  Number of AOs       : 2
INFO:QMCTorch|  Number of MOs       : 2
INFO:QMCTorch|  SCF Energy          : -1.066 Hartree
```

The use of GPU acceleration has been streamlined in QMCTorch, the only modification you need to do on your code is to specify `cuda=True` in the declaration of the wave function and sampler, this will automatically port all the necesaary tensors to the GPU and offload all the corresponding operation there.

[9]:
```
if torch.cuda.is_available():
    wf = SlaterJastrow(mol, cuda=True)
    sampler = Metropolis(nwalkers=100, nstep=500, step_size=0.25,
                         nelec=wf.nelec, ndim=wf.ndim,
                         init=mol.domain('atomic'),
                         move={'type': 'all-elec', 'proba': 'normal'},
                         cuda=True)
else:
    print('CUDA not available, install torch with cuda support to proceed')
```

```
CUDA not available, install torch with cuda support to proceed
```

CHAPTER 10

Correlation and Blocking

One important part of the sampling is to estimate the correlation between the different sampling points. To this end let's import the following modules

[8]:
```
from qmctorch.scf import Molecule
from qmctorch.wavefunction import SlaterJastrow
from qmctorch.sampler import Metropolis
from qmctorch.solver import Solver
from qmctorch.utils import set_torch_double_precision
from qmctorch.utils import blocking, plot_blocking_energy
from qmctorch.utils import plot_correlation_coefficient, plot_integrated_
↪autocorrelation_time
```

10.1 Setting up the system

Let's create a simple H2 molecule and a Slater Jastrow wave fuction to demonstrate the correlation properties of the sampling

[6]:
```
set_torch_double_precision()
mol = Molecule(atom = 'H 0. 0. 0; H 0. 0. 1.', unit='bohr', redo_scf=True)
```

```
INFO:QMCTorch|
INFO:QMCTorch| SCF Calculation
INFO:QMCTorch|  Removing H2_pyscf_sto-3g.hdf5 and redo SCF calculations
INFO:QMCTorch|  Running scf  calculation
converged SCF energy = -1.06599946214331
INFO:QMCTorch|  Molecule name       : H2
INFO:QMCTorch|  Number of electrons : 2
INFO:QMCTorch|  SCF calculator      : pyscf
INFO:QMCTorch|  Basis set           : sto-3g
INFO:QMCTorch|  SCF                 : HF
INFO:QMCTorch|  Number of AOs       : 2
```

(continues on next page)

(continued from previous page)

```
INFO:QMCTorch|  Number of MOs       : 2
INFO:QMCTorch|  SCF Energy          : -1.066 Hartree
```

[7]:
```
wf = SlaterJastrow(mol, configs='ground_state')
```

```
INFO:QMCTorch|
INFO:QMCTorch| Wave Function
INFO:QMCTorch|  Jastrow factor      : True
INFO:QMCTorch|  Jastrow kernel      : PadeJastrowKernel
INFO:QMCTorch|  Highest MO included : 2
INFO:QMCTorch|  Configurations      : ground_state
INFO:QMCTorch|  Number of confs     : 1
INFO:QMCTorch|  Kinetic energy      : jacobi
INFO:QMCTorch|  Number var  param   : 18
INFO:QMCTorch|  Cuda support        : False
```

We can also define a simple Metropolis sampler:

[5]:
```
sampler = Metropolis(nwalkers=100, nstep=500, step_size=0.25,
                     nelec=wf.nelec, ndim=wf.ndim,
                     init=mol.domain('normal'),
                     ntherm=0, ndecor=1,
                     move={'type': 'all-elec', 'proba': 'normal'})
```

```
INFO:QMCTorch|
INFO:QMCTorch| Monte-Carlo Sampler
INFO:QMCTorch|  Number of walkers   : 100
INFO:QMCTorch|  Number of steps     : 500
INFO:QMCTorch|  Step size           : 0.25
INFO:QMCTorch|  Thermalization steps: 0
INFO:QMCTorch|  Decorelation steps  : 1
INFO:QMCTorch|  Walkers init pos    : normal
INFO:QMCTorch|  Move type           : all-elec
INFO:QMCTorch|  Move proba          : normal
```

note that by setting `nthemr=0` and `ndecor=1`, we record all the walker positions along the trajectory.

We can then define the solver:

[9]:
```
solver = Solver(wf=wf, sampler=sampler)
```

```
INFO:QMCTorch|
INFO:QMCTorch| QMC Solver
INFO:QMCTorch|  WaveFunction        : SlaterJastrow
INFO:QMCTorch|  Sampler             : Metropolis
```

10.2 Correlation coefficient

The simplest way to estimate the decorelation time is to compute the autocorrelation coefficient of the local energy. To this end we must first record the sampling trajectory, i.e. the values of the local energies along the path of the walkers:

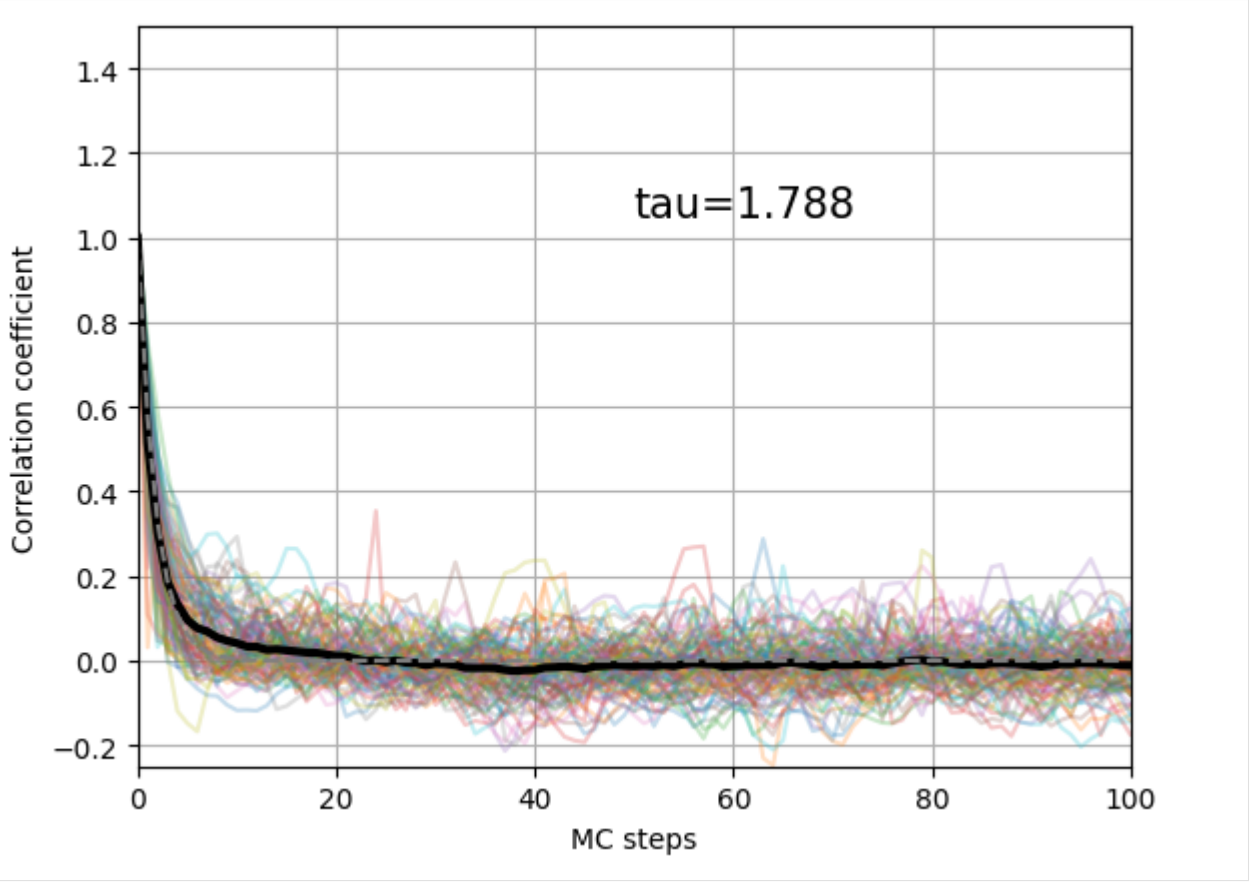
[10]:
```
pos = solver.sampler(solver.wf.pdf)
obs = solver.sampling_traj(pos)
```

```
INFO:QMCTorch|  Sampling: 100%|| 500/500 [00:46<00:00, 10.68it/s]
```

```
INFO:QMCTorch|   Acceptance rate      : 62.55 %
INFO:QMCTorch|   Timing statistics    : 10.68 steps/sec.
INFO:QMCTorch|   Total Time           : 46.83 sec.
INFO:QMCTorch|
INFO:QMCTorch|  Sampling trajectory
INFO:QMCTorch|  Energy  : 100%|| 500/500 [01:43<00:00,  4.85it/s]
```

We can then plot the correlation coefficient with:

[11]:
```
rho, tau = plot_correlation_coefficient(obs.local_energy)
```

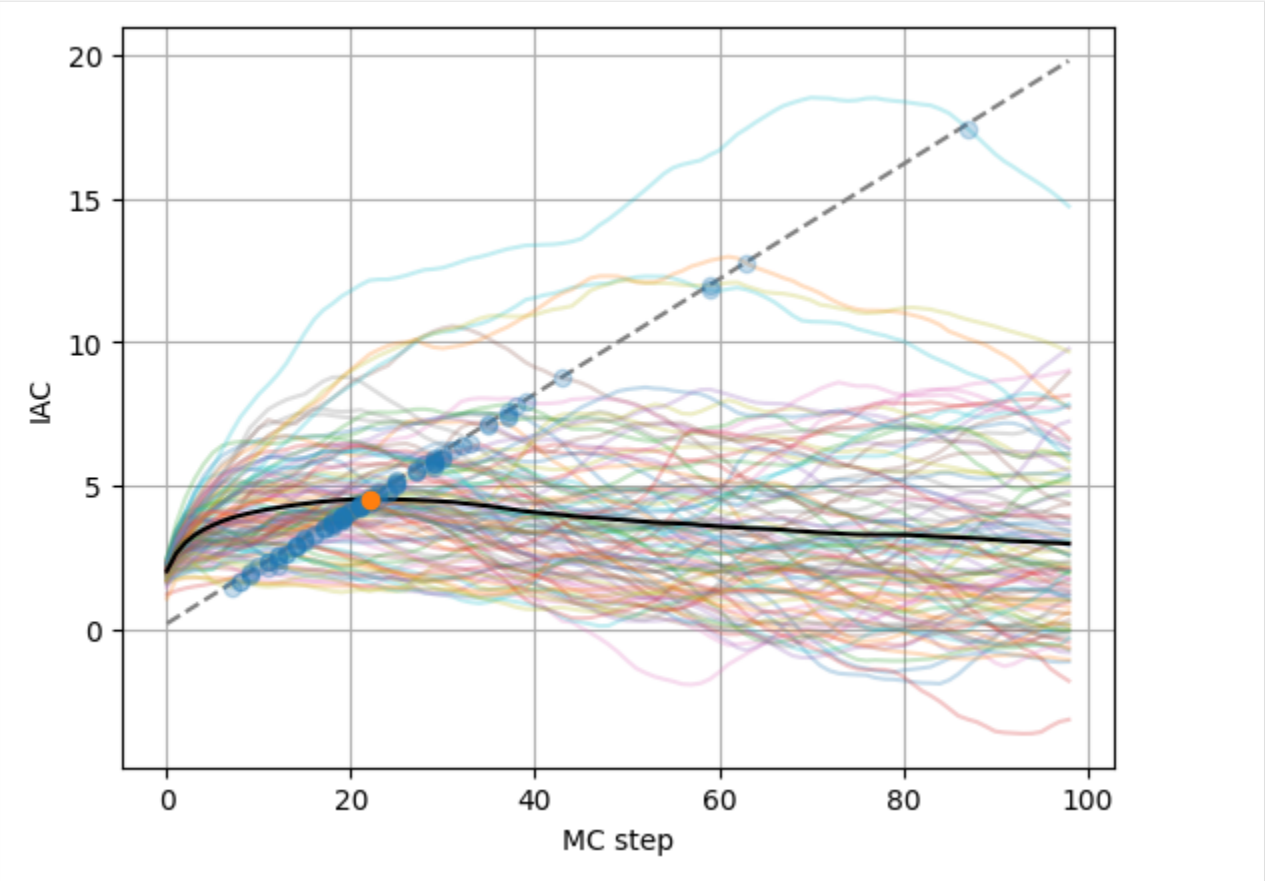


On this picture is represented the autocorrelation coefficient of the local energy of all the walkers (transparent colorful line) and the average of the autocorrelation coefficient (thick black line). This mean value is fitted with an exponential decay to obtain the autocorrelation time that is here equal to 1.79 MCs.

10.3 Integrated autocorrelation time

Another way to estimate the correlation time is to compute the integrated autocorrelation time. This can be done with

[12]:
```
plot_integrated_autocorrelation_time(obs.local_energy)
```

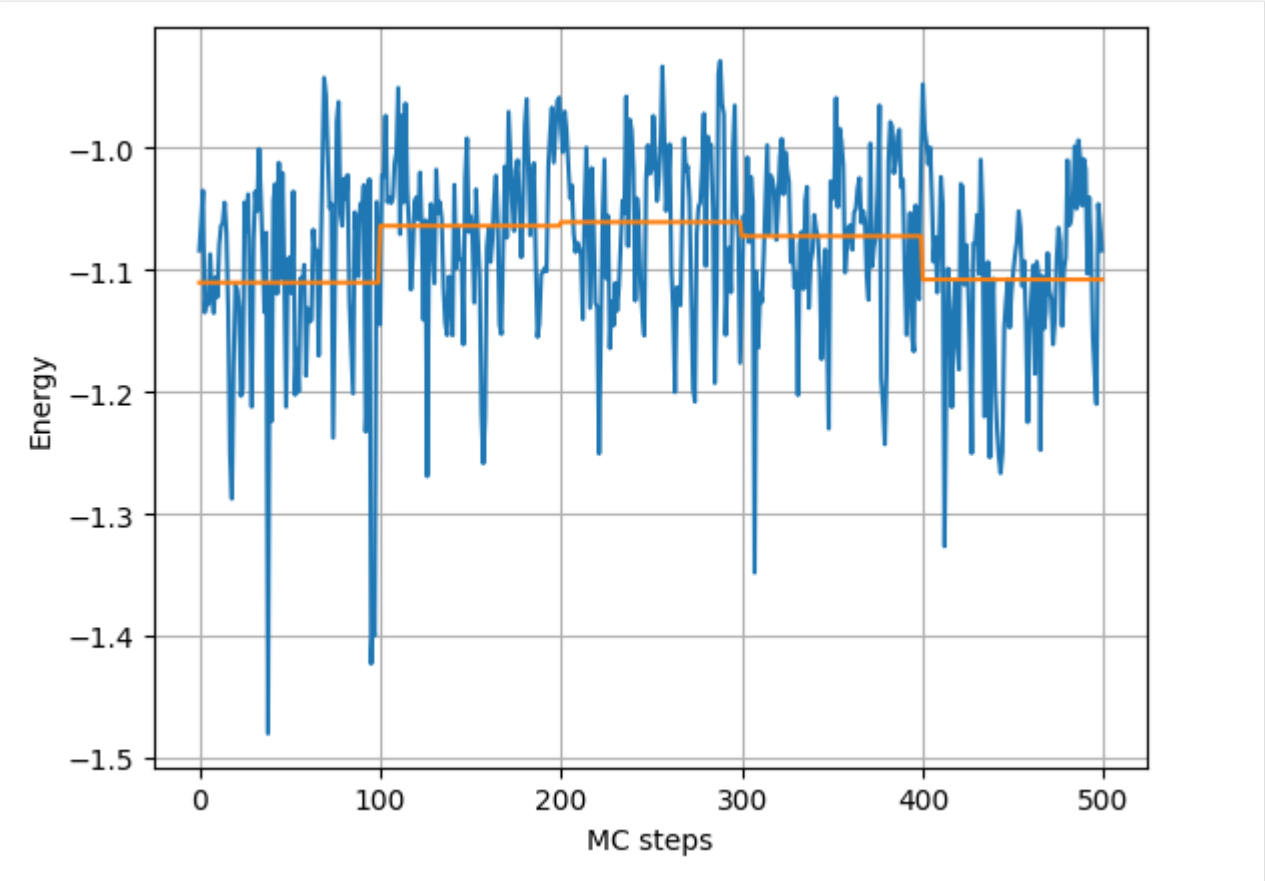


A conservative estimate of the correlation time can be obtain when the iac cross the dashed line, leading here to a value of about 20 steps.

10.4 Energy blocking

It is also common practice to use blocking of the local energy values to reduce the variance. This can easily be done with

[13]:
```
eb = plot_blocking_energy(obs.local_energy, block_size=100, walkers='mean')
```



That shows the raw and blocked values of the mean local energy values.

CHAPTER 11

Creating your own Jastrow Factor

We present here how to create your own electron-electron Jastrow factor and use it in QMCTorch. During the import you must import the base class of the electron-electron Jastrow. We aso create a H2 molecule

[6]:
```
from qmctorch.scf import Molecule
from qmctorch.wavefunction import SlaterJastrow
from qmctorch.wavefunction.jastrows.elec_elec.kernels import␣
↪JastrowKernelElectronElectronBase
mol = Molecule(atom='H 0. 0. 0; H 0. 0. 1.', unit='bohr', redo_scf=True)
```

```
INFO:QMCTorch|
INFO:QMCTorch| SCF Calculation
INFO:QMCTorch|  Removing H2_pyscf_sto-3g.hdf5 and redo SCF calculations
INFO:QMCTorch|  Running scf  calculation
converged SCF energy = -1.06599946214331
INFO:QMCTorch|  Molecule name       : H2
INFO:QMCTorch|  Number of electrons : 2
INFO:QMCTorch|  SCF calculator      : pyscf
INFO:QMCTorch|  Basis set           : sto-3g
INFO:QMCTorch|  SCF                 : HF
INFO:QMCTorch|  Number of AOs       : 2
INFO:QMCTorch|  Number of MOs       : 2
INFO:QMCTorch|  SCF Energy          : -1.066 Hartree
```

We can then use this base class to create a new Jastrow Factor. This is done in the same way one would create a new neural network layer in pytorch.

[2]:
```
from torch import nn
class MyJastrow(JastrowKernelElectronElectronBase):
    def __init__(self, nup, ndown, cuda, size=16):
        super().__init__(nup, ndown, cuda)
        self.fc1 = nn.Linear(1, size, bias=False)
        self.fc2 = nn.Linear(size, 1, bias=False)
    def forward(self, x):
        nbatch, npair = x.shape
        x = x.reshape(-1,1)
```

(continues on next page)

(continued from previous page)

```
        x = self.fc2(self.fc1(x))
        return x.reshape(nbatch, npair)
```

As seen above the prototype of the class constructor must be:

`def __init__(self, nup, ndown, cuda, **kwargs)`

The list of keyword argument can contain any pairs such as `size=16`.

This Jastrow use two fully connected layers. The size of the hidden layer is here controlled by a keyword argument `size` whose defauilt value is 16 It is important to note that the calculation of the first and second derivative of the jastrow kernel wrt the electronic positions are then done via automatic differentiation as implemented in the `JastrowKernelElectronElectronBase` class. Hence there is no need to derive and implement these derivatives. However it is necessary that the `forward` function, which takes as input a `torch.tensor` of dimension `[Nbatch, Npair]` first reshape this tensor to `[Nbatch*Npair,1]`, then applies the transformation on this tensor and finally reshape the output tensor to `[Nbatch, Npair]`.

To use this new Jastrow in the `SlaterJastrow` wave function ansatz we simply pass the class name as argument of the `jastrow_kernel` keyword argument. It is also possible to specify the values of the keyword argument `size` with the `jastrow_kernel_kwargs`. As seen below the pair of keyword argument and its value is passed as a python dictionary :

[5]:
```
wf = SlaterJastrow(mol, jastrow_kernel=MyJastrow, jastrow_kernel_kwargs={'size' : 64})
```

```
INFO:QMCTorch|
INFO:QMCTorch| SCF Calculation
INFO:QMCTorch|  Running scf  calculation
converged SCF energy = -1.06599946214331
INFO:QMCTorch|  Molecule name       : H2
INFO:QMCTorch|  Number of electrons : 2
INFO:QMCTorch|  SCF calculator      : pyscf
INFO:QMCTorch|  Basis set           : sto-3g
INFO:QMCTorch|  SCF                 : HF
INFO:QMCTorch|  Number of AOs       : 2
INFO:QMCTorch|  Number of MOs       : 2
INFO:QMCTorch|  SCF Energy          : -1.066 Hartree
INFO:QMCTorch|
INFO:QMCTorch| Wave Function
INFO:QMCTorch|  Jastrow factor      : True
INFO:QMCTorch|  Jastrow kernel      : MyJastrow
INFO:QMCTorch|  Highest MO included : 2
INFO:QMCTorch|  Configurations      : ground_state
INFO:QMCTorch|  Number of confs     : 1
INFO:QMCTorch|  Kinetic energy      : jacobi
INFO:QMCTorch|  Number var  param   : 145
INFO:QMCTorch|  Cuda support        : False
```

[ ]:

CHAPTER 12

Creating your own Backflow transformation

We present here how to create your own backflow transformation and use it in QMCTorch. During the import you must import the base class of the backflow kernel. We aso create a H2 molecule

[4]:
```
import torch
from qmctorch.scf import Molecule
from qmctorch.wavefunction import SlaterJastrowBackFlow
from qmctorch.wavefunction.orbitals.backflow.kernels import BackFlowKernelBase
mol = Molecule(atom='H 0. 0. 0; H 0. 0. 1.', unit='bohr', redo_scf=True)
```

```
INFO:QMCTorch|
INFO:QMCTorch| SCF Calculation
INFO:QMCTorch|  Running scf  calculation
converged SCF energy = -1.06599946214331
INFO:QMCTorch|  Molecule name       : H2
INFO:QMCTorch|  Number of electrons : 2
INFO:QMCTorch|  SCF calculator      : pyscf
INFO:QMCTorch|  Basis set           : sto-3g
INFO:QMCTorch|  SCF                 : HF
INFO:QMCTorch|  Number of AOs       : 2
INFO:QMCTorch|  Number of MOs       : 2
INFO:QMCTorch|  SCF Energy          : -1.066 Hartree
```

We can then use this base class to create a new backflow transformation kernel. This is done in the same way one would create a new neural network layer in pytorch

[5]:
```
from torch import nn
class MyBackflow(BackFlowKernelBase):
    def __init__(self, mol, cuda, size=16):
        super().__init__(mol, cuda)
        self.fc1 = nn.Linear(1, size, bias=False)
        self.fc2 = nn.Linear(size, 1, bias=False)
    def forward(self, x):
        original_shape = x.shape
        x = x.reshape(-1,1)
```

(continues on next page)

(continued from previous page)

```
        x = self.fc2(self.fc1(x))
        return x.reshape(*original_shape)
```

This backflow transformation consists of two fully connected layers. The calculation of the first and second derivative are then done via automatic differentiation as implemented in the `BackFlowKernelBase` class. To use this new kernel in the `SlaterJastrowBackFlow` wave function ansatz we simply pass the class name as argument of the `backflow_kernel` keyword argument :

[6]:
```
wf = SlaterJastrowBackFlow(mol,
                   backflow_kernel=MyBackflow,
                   backflow_kernel_kwargs={'size' : 64})
```

```
INFO:QMCTorch|
INFO:QMCTorch| Wave Function
INFO:QMCTorch|  Jastrow factor      : True
INFO:QMCTorch|  Jastrow kernel      : PadeJastrowKernel
INFO:QMCTorch|  Highest MO included : 2
INFO:QMCTorch|  Configurations      : ground_state
INFO:QMCTorch|  Number of confs     : 1
INFO:QMCTorch|  Kinetic energy      : jacobi
INFO:QMCTorch|  Number var  param   : 146
INFO:QMCTorch|  Cuda support        : False
```

[7]:
```
pos = torch.rand(10, wf.nelec*3)
print(wf(pos))
```

```
tensor([[0.1134],
        [0.1509],
        [0.1096],
        [0.1093],
        [0.2632],
        [0.1523],
        [0.1253],
        [0.1424],
        [0.1324],
        [0.0665]], grad_fn=<MulBackward0>)
```

[ ]:
```

```

CHAPTER 13

Multi CPUS/GPUs support with Horovod

Warning The use mutli-GPU is under developpement and hasn't been thoroughly tested yet. Proceed with caution !

QMC simulations can easily be parallelized by using multiple ressources to sample the wave function. Each walker is indenpendent of the other ones and therefore multiple compute node can be used in parallel to obtain more samples. Each node can alsu use GPUs is they are available. We demonstrate here how to use the library `Horovod` (https://github.com/horovod/horovod) to leverage large compute ressources for QMC.

Let's first create a simple system

[1]:
```
import torch
from torch import optim
from qmctorch.scf import Molecule
from qmctorch.wavefunction import SlaterJastrow
from qmctorch.sampler import Metropolis
from qmctorch.utils import (plot_energy, plot_data)
from qmctorch.utils import set_torch_double_precision
set_torch_double_precision()
mol = Molecule(atom='H 0. 0. 0; H 0. 0. 1.', unit='bohr', redo_scf=True)
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
/tmp/ipykernel_633/493514665.py in <module>
----> 1 import torch
      2 from torch import optim
      3 from qmctorch.scf import Molecule
      4 from qmctorch.wavefunction import SlaterJastrow
      5 from qmctorch.sampler import Metropolis

ModuleNotFoundError: No module named 'torch'
```

Let's see if GPUs are available

[2]:
```
use_gpu = torch.cuda.is_available()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_633/3587241540.py in <module>
----> 1 use_gpu = torch.cuda.is_available()

NameError: name 'torch' is not defined
```

[3]:
```
wf = SlaterJastrow(mol, cuda=use_gpu).gto2sto()
sampler = Metropolis(nwalkers=100, nstep=500, step_size=0.25,
                     nelec=wf.nelec, ndim=wf.ndim,
                     init=mol.domain('atomic'),
                     move={'type': 'all-elec', 'proba': 'normal'},
                     cuda=use_gpu)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_633/3194883153.py in <module>
----> 1 wf = SlaterJastrow(mol, cuda=use_gpu).gto2sto()
      2 sampler = Metropolis(nwalkers=100, nstep=500, step_size=0.25,
      3                      nelec=wf.nelec, ndim=wf.ndim,
      4                      init=mol.domain('atomic'),
      5                      move={'type': 'all-elec', 'proba': 'normal'},

NameError: name 'SlaterJastrow' is not defined
```

[4]:
```
lr_dict = [{'params': wf.jastrow.parameters(), 'lr': 3E-3},
           {'params': wf.ao.parameters(), 'lr': 1E-6},
           {'params': wf.mo.parameters(), 'lr': 1E-3},
           {'params': wf.fc.parameters(), 'lr': 2E-3}]
opt = optim.Adam(lr_dict, lr=1E-3)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_633/358553228.py in <module>
----> 1 lr_dict = [{'params': wf.jastrow.parameters(), 'lr': 3E-3},
      2            {'params': wf.ao.parameters(), 'lr': 1E-6},
      3            {'params': wf.mo.parameters(), 'lr': 1E-3},
      4            {'params': wf.fc.parameters(), 'lr': 2E-3}]
      5 opt = optim.Adam(lr_dict, lr=1E-3)

NameError: name 'wf' is not defined
```

A dedicated QMCTorch Solver has been developped to handle multiple GPU. To use this solver simply import it and use is as the normal solver and only a few modifications are required to use horovod :

[5]:
```
import horovod.torch as hvd
from qmctorch.solver import SolverMPI

hvd.init()
if torch.cuda.is_available():
    torch.cuda.set_device(hvd.rank())

solver = SolverMPI(wf=wf, sampler=sampler,
                                  optimizer=opt,
                                  rank=hvd.rank())
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
```

(continues on next page)

(continued from previous page)

```
/tmp/ipykernel_633/263085700.py in <module>
----> 1 import horovod.torch as hvd
      2 from qmctorch.solver import SolverMPI
      3
      4 hvd.init()
      5 if torch.cuda.is_available():

ModuleNotFoundError: No module named 'horovod'
```

[6]:
```
solver.configure(track=['local_energy'], freeze=['ao', 'mo'],
                 loss='energy', grad='auto',
                 ortho_mo=False, clip_loss=False,
                 resampling={'mode': 'update',
                             'resample_every': 1,
                             'nstep_update': 50})

# optimize the wave function
obs = solver.run(5)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_633/214530952.py in <module>
----> 1 solver.configure(track=['local_energy'], freeze=['ao', 'mo'],
      2                  loss='energy', grad='auto',
      3                  ortho_mo=False, clip_loss=False,
      4                  resampling={'mode': 'update',
      5                              'resample_every': 1,

NameError: name 'solver' is not defined
```

As you can see some classes need the rank of the process when they are defined. This is simply to insure that only the master process generates the HDF5 files containing the information relative to the calculation.

13.1 Running parallel calculations

It is currently difficult to use Horovod on mutliple node through a jupyter notebook. To do so, one should have a python file with all the code and execute the code with the following command

```
horovodrun -np 2 python <example>.py
```

See the horovod documentation for more details : https://github.com/horovod/horovod

This solver distribute the `Nw` walkers over the `Np` process . For example specifying 2000 walkers and using 4 process will lead to each process using only 500 walkers. During the optimizaiton of the wavefunction each process will compute the gradients of the variational parameter using their local 500 walkers. The gradients are then averaged over all the processes before the optimization step takes place. This data parallel model has been greatly succesfull in machine learning applications (http://jmlr.org/papers/volume20/18-789/18-789.pdf)

A complete example can found in `qmctorch/docs/example/horovod/h2.py`

CHAPTER 14

Subpackages

14.1 qmctorch.sampler package

14.1.1 Submodules

qmctorch.sampler.generalized_metropolis module

class qmctorch.sampler.generalized_metropolis.**GeneralizedMetropolis**(*nwalkers=100*, *nstep=1000*, *step_size=3*, *ntherm=-1*, *ndecor=1*, *nelec=1*, *ndim=1*, *init={'max': 5, 'min': -5, 'type': 'uniform'}*, *cuda=False*)

Bases: *qmctorch.sampler.sampler_base.SamplerBase*

Generalized Metropolis Hasting sampler

Parameters

- **nwalkers** (*int, optional*) – number of walkers. Defaults to 100.
- **nstep** (*int, optional*) – number of steps. Defaults to 1000.
- **step_size** (*int, optional*) – size of the steps. Defaults to 3.
- **ntherm** (*int, optional*) – number of steps for thermalization. Defaults to -1.

- **ndecor** (*int, optional*) – number of steps for decorelation. Defaults to 1.
- **nelec** (*int, optional*) – number of electron. Defaults to 1.
- **ndim** (*int, optional*) – number of dimensions. Defaults to 1.
- **init** (*dict, optional*) – method to initialize the walkers. Defaults to {'type': 'uniform', 'min': -5, 'max': 5}.
- **cuda** (*bool, optional*) – use cuda. Defaults to False.

move(*drift*)
Move electron one at a time in a vectorized way.

Parameters **drift** (*torch.tensor*) – drift velocity of the walkers

Returns new positions of the walkers

Return type torch.tensor

trans(*xf*, *xi*, *drifti*)
transform the positions

Parameters

- **xf** (*[type]*) – [description]
- **xi** (*[type]*) – [description]
- **drifti** (*[type]*) – [description]

Returns [description]

Return type [type]

get_drift(*pdf*, *x*)
Compute the drift velocity

Parameters

- **pdf** (*callable*) – function that returns the density
- **x** (*torch.tensor*) – positions of the walkers

Returns drift velocity

Return type torch.tensor

qmctorch.sampler.hamiltonian module

class qmctorch.sampler.hamiltonian.**Hamiltonian**(*nwalkers: int = 100*, *nstep: int = 100*, *step_size: float = 0.2*, *L: int = 10*, *ntherm: int = -1*, *ndecor: int = 1*, *nelec: int = 1*, *ndim: int = 3*, *init: Dict[KT, VT] = {'max': 5, 'min': -5}*, *cuda: bool = False*)
Bases: *qmctorch.sampler.sampler_base.SamplerBase*

Hamiltonian Monte Carlo Sampler.

Parameters

- **nwalkers** (*int, optional*) – Number of walkers. Defaults to 100.
- **nstep** (*int, optional*) – Number of steps. Defaults to 100.

- **step_size** (*int, optional*) – length of the step. Defaults to 0.2.
- **L** (*int, optional*) – length of the trajectory . Defaults to 10.
- **nelec** (*int, optional*) – total number of electrons. Defaults to 1.
- **ntherm** (*int, optional*) – number of mc step to thermalize. Defaults to -1, i.e. keep only last position
- **ndecor** (*int, optional*) – number of mc step for decorrelation. Defaults to 1.
- **ndim** (*int, optional*) – total number of dimension. Defaults to 3.
- **init** (*dict, optional*) – method to init the positions of the walkers. See Molecule.domain()
- **cuda** (*bool, optional*) – turn CUDA ON/OFF. Defaults to False.

static get_grad(*func*, *inp*)
get the gradient of the pdf using autograd

Parameters

- **func** (*callable*) – function to compute the pdf
- **inp** (*torch.tensor*) – input of the function

Returns gradients of the wavefunction

Return type torch.tensor

static log_func(*func*)
Compute the negative log of a function

Parameters **func** (*callable*) – input function

Returns negative log of the function

Return type callable

qmctorch.sampler.metropolis module

class qmctorch.sampler.metropolis.**Metropolis**(*nwalkers: int = 100, nstep: int = 1000, step_size: float = 0.2, ntherm: int = -1, ndecor: int = 1, nelec: int = 1, ndim: int = 3, init: Dict[KT, VT] = {'max': 5, 'min': -5}, move: Dict[KT, VT] = {'proba': 'normal', 'type': 'all-elec'}, cuda: bool = False*)
Bases: *qmctorch.sampler.sampler_base.SamplerBase*

Metropolis Hasting generator

Parameters

- **nwalkers** (*int, optional*) – Number of walkers. Defaults to 100.
- **nstep** (*int, optional*) – Number of steps. Defaults to 1000.
- **step_size** (*int, optional*) – length of the step. Defaults to 0.2.
- **nelec** (*int, optional*) – total number of electrons. Defaults to 1.
- **ntherm** (*int, optional*) – number of mc step to thermalize. Defaults to -1, i.e. keep only the last position

- **ndecor** (*int, optional*) – number of mc step for decorelation. Defauts to 1.
- **ndim** (*int, optional*) – total number of dimension. Defaults to 3.
- **init** (*dict, optional*) – method to init the positions of the walkers. See Molecule.domain()
- **move** (*dict, optional*) – method to move the electrons. default('all-elec','normal')

  'type':

  'one-elec': move a single electron per iteration

  'all-elec': move all electrons at the same time

  'all-elec-iter': move all electrons by iterating through single elec moves

  'proba' : 'uniform': uniform in a cube

  'normal': gussian in a sphere

- **cuda** (*bool, optional*) – turn CUDA ON/OFF. Defaults to False.

Examples::

```
>>> mol = Molecule('h2.xyz')
>>> wf = SlaterJastrow(mol)
>>> sampler = Metropolis(nwalkers=100, nelec=wf.nelec)
>>> pos = sampler(wf.pdf)
```

log_data()
log data about the sampler.

configure_move(*move: Dict[KT, VT]*)
Configure the electron moves

Parameters **move** (*dict, optional*) – method to move the electrons. default('all-elec','normal')

'type':

'one-elec': move a single electron per iteration

'all-elec': move all electrons at the same time

'all-elec-iter': move all electrons by iterating through single elec moves

'proba' : 'uniform': uniform ina cube

'normal': gussian in a sphere

Raises ValueError – If moves are not recognized

move(*pdf: Callable, id_elec: int*) → <sphinx.ext.autodoc.importer._MockObject object at 0x7f95a2ad1810>
Move electron one at a time in a vectorized way.

Parameters

- **pdf** (*callable*) – function to sample
- **id_elec** (*int*) – index f the electron to move

Returns new positions of the walkers

Return type torch.tensor

qmctorch.sampler.sampler_base module

class qmctorch.sampler.sampler_base.**SamplerBase**(*nwalkers*, *nstep*, *step_size*, *ntherm*, *ndecor*, *nelec*, *ndim*, *init*, *cuda*)

Bases: object

Base class for the sampler

Parameters

- **nwalkers** (*int*) – number of walkers
- **nstep** (*int*) – number of MC steps
- **step_size** (*float*) – size of the steps in bohr
- **ntherm** (*int*) – number of MC steps to thermalize
- **ndecor** (*int*) – unmber of MC steps to decorellate
- **nelec** (*int*) – number of electrons in the system
- **ndim** (*int*) – number of cartesian dimension
- **init** (*dict*) – method to initialize the walkers
- **cuda** (*[type]*) – [description]

get_sampling_size()
evaluate the number of sampling point we'll have.

qmctorch.sampler.walkers module

class qmctorch.sampler.walkers.**Walkers**(*nwalkers: int = 100*, *nelec: int = 1*, *ndim: int = 3*, *init: Optional[Dict[KT, VT]] = None*, *cuda: bool = False*)

Bases: object

Creates Walkers for the sampler.

Parameters

- **nwalkers** (*int, optional*) – Number of walkers. Defaults to 100.
- **nelec** (*int, optional*) – number of electron. Defaults to 1.
- **ndim** (*int, optional*) – Number of dimensions. Defaults to 3.
- **init** (*dict, optional*) – method to initialize the walkers. Defaults to None. (see Molecule.domain())
- **cuda** (*bool, optional*) – turn cuda ON/OFF. Defaults to False

initialize(*pos: Union[None, <sphinx.ext.autodoc.importer._MockObject object at 0x7f95a3c59ed0>] = None*)
Initalize the position of the walkers

Parameters

- **method** (*str, optional*) – how to initialize the positions. Defaults to 'uniform'.
- **pos** (*[type], optional*) – existing position of the walkers. Defaults to None.

Raises ValueError – if the method is not recognized

14.1.2 Module contents

class qmctorch.sampler.**SamplerBase**(*nwalkers*, *nstep*, *step_size*, *ntherm*, *ndecor*, *nelec*, *ndim*, *init*, *cuda*)

Bases: object

Base class for the sampler

Parameters

- **nwalkers** (*int*) – number of walkers
- **nstep** (*int*) – number of MC steps
- **step_size** (*float*) – size of the steps in bohr
- **ntherm** (*int*) – number of MC steps to thermalize
- **ndecor** (*int*) – unmber of MC steps to decorellate
- **nelec** (*int*) – number of electrons in the system
- **ndim** (*int*) – number of cartesian dimension
- **init** (*dict*) – method to initialize the walkers
- **cuda** (*[type]*) – [description]

get_sampling_size()
evaluate the number of sampling point we'll have.

class qmctorch.sampler.**Metropolis**(*nwalkers: int = 100*, *nstep: int = 1000*, *step_size: float = 0.2*, *ntherm: int = -1*, *ndecor: int = 1*, *nelec: int = 1*, *ndim: int = 3*, *init: Dict[KT, VT] = {'max': 5, 'min': -5}*, *move: Dict[KT, VT] = {'proba': 'normal', 'type': 'all-elec'}*, *cuda: bool = False*)

Bases: *qmctorch.sampler.sampler_base.SamplerBase*

Metropolis Hasting generator

Parameters

- **nwalkers** (*int, optional*) – Number of walkers. Defaults to 100.
- **nstep** (*int, optional*) – Number of steps. Defaults to 1000.
- **step_size** (*int, optional*) – length of the step. Defaults to 0.2.
- **nelec** (*int, optional*) – total number of electrons. Defaults to 1.
- **ntherm** (*int, optional*) – number of mc step to thermalize. Defaults to -1, i.e. keep only the last position
- **ndecor** (*int, optional*) – number of mc step for decorelation. Defauts to 1.
- **ndim** (*int, optional*) – total number of dimension. Defaults to 3.
- **init** (*dict, optional*) – method to init the positions of the walkers. See Molecule.domain()
- **move** (*dict, optional*) – method to move the electrons. default('all-elec','normal')

  'type':

'one-elec': move a single electron per iteration

'all-elec': move all electrons at the same time

'all-elec-iter': move all electrons by iterating through single elec moves

'proba' : 'uniform': uniform in a cube

'normal': gussian in a sphere

- **cuda** (*bool, optional*) – turn CUDA ON/OFF. Defaults to False.

Examples::

```
>>> mol = Molecule('h2.xyz')
>>> wf = SlaterJastrow(mol)
>>> sampler = Metropolis(nwalkers=100, nelec=wf.nelec)
>>> pos = sampler(wf.pdf)
```

log_data()
log data about the sampler.

configure_move(*move: Dict[KT, VT]*)
Configure the electron moves

Parameters **move** (*dict, optional*) – method to move the electrons. default('all-elec','normal')

'type':

'one-elec': move a single electron per iteration

'all-elec': move all electrons at the same time

'all-elec-iter': move all electrons by iterating through single elec moves

'proba' : 'uniform': uniform ina cube

'normal': gussian in a sphere

Raises ValueError – If moves are not recognized

move(*pdf: Callable, id_elec: int*) → <sphinx.ext.autodoc.importer._MockObject object at 0x7f95a2ad1810>
Move electron one at a time in a vectorized way.

Parameters

- **pdf** (*callable*) – function to sample
- **id_elec** (*int*) – index f the electron to move

Returns new positions of the walkers

Return type torch.tensor

class qmctorch.sampler.**Hamiltonian**(*nwalkers: int = 100, nstep: int = 100, step_size: float = 0.2, L: int = 10, ntherm: int = -1, ndecor: int = 1, nelec: int = 1, ndim: int = 3, init: Dict[KT, VT] = {'max': 5, 'min': -5}, cuda: bool = False*)
Bases: *qmctorch.sampler.sampler_base.SamplerBase*

Hamiltonian Monte Carlo Sampler.

Parameters

- **nwalkers** (*int, optional*) – Number of walkers. Defaults to 100.
- **nstep** (*int, optional*) – Number of steps. Defaults to 100.
- **step_size** (*int, optional*) – length of the step. Defaults to 0.2.
- **L** (*int, optional*) – length of the trajectory . Defaults to 10.
- **nelec** (*int, optional*) – total number of electrons. Defaults to 1.
- **ntherm** (*int, optional*) – number of mc step to thermalize. Defaults to -1, i.e. keep only last position
- **ndecor** (*int, optional*) – number of mc step for decorrelation. Defaults to 1.
- **ndim** (*int, optional*) – total number of dimension. Defaults to 3.
- **init** (*dict, optional*) – method to init the positions of the walkers. See Molecule.domain()
- **cuda** (*bool, optional*) – turn CUDA ON/OFF. Defaults to False.

static get_grad(*func*, *inp*)
get the gradient of the pdf using autograd

Parameters

- **func** (*callable*) – function to compute the pdf
- **inp** (*torch.tensor*) – input of the function

Returns gradients of the wavefunction

Return type torch.tensor

static log_func(*func*)
Compute the negative log of a function

Parameters **func** (*callable*) – input function

Returns negative log of the function

Return type callable

class qmctorch.sampler.**GeneralizedMetropolis**(*nwalkers=100*, *nstep=1000*, *step_size=3*, *ntherm=-1*, *ndecor=1*, *nelec=1*, *ndim=1*, *init={'max': 5, 'min': -5, 'type': 'uniform'}*, *cuda=False*)

Bases: *qmctorch.sampler.sampler_base.SamplerBase*

Generalized Metropolis Hasting sampler

Parameters

- **nwalkers** (*int, optional*) – number of walkers. Defaults to 100.
- **nstep** (*int, optional*) – number of steps. Defaults to 1000.
- **step_size** (*int, optional*) – size of the steps. Defaults to 3.
- **ntherm** (*int, optional*) – number of steps for thermalization. Defaults to -1.
- **ndecor** (*int, optional*) – number of steps for decorelation. Defaults to 1.
- **nelec** (*int, optional*) – number of electron. Defaults to 1.
- **ndim** (*int, optional*) – number of dimensions. Defaults to 1.

- **init** (*dict, optional*) – method to initialize the walkers. Defaults to {'type': 'uniform', 'min': -5, 'max': 5}.
- **cuda** (*bool, optional*) – use cuda. Defaults to False.

move(*drift*)
Move electron one at a time in a vectorized way.

Parameters **drift** (*torch.tensor*) – drift velocity of the walkers

Returns new positions of the walkers

Return type torch.tensor

trans(*xf*, *xi*, *drifti*)
transform the positions

Parameters

- **xf** (*[type]*) – [description]
- **xi** (*[type]*) – [description]
- **drifti** (*[type]*) – [description]

Returns [description]

Return type [type]

get_drift(*pdf*, *x*)
Compute the drift velocity

Parameters

- **pdf** (*callable*) – function that returns the density
- **x** (*torch.tensor*) – positions of the walkers

Returns drift velocity

Return type torch.tensor

14.2 qmctorch.scf package

14.2.1 Subpackages

qmctorch.scf.calculator package

Submodules

qmctorch.scf.calculator.adf module

class qmctorch.scf.calculator.adf.**CalculatorADF**(*atoms*, *atom_coords*, *basis*, *scf*, *units*, *molname*, *savefile*)
Bases: *qmctorch.scf.calculator.calculator_base.CalculatorBase*

run()
Run the calculation using ADF.

init_plams()
Init PLAMS.

get_plams_molecule()
Returns a plams molecule object.

get_plams_settings()
Returns a plams setting object.

get_basis_data(*kffile*)
Save the basis information needed to compute the AO values.

static read_array(*kf*, *section*, *name*)
read a data from the kf file

Parameters

- **kf** (*file handle*) – kf file
- **section** (*str*) – name of the section
- **name** (*str*) – name of the property

Returns data

Return type np.data

class qmctorch.scf.calculator.adf.**CalculatorADF2019**(*atoms*, *atom_coords*, *basis*, *scf*, *units*, *molname*, *savefile*)
Bases: *qmctorch.scf.calculator.adf.CalculatorADF*

get_plams_molecule()
Returns a plams molecule object.

get_plams_settings()
Returns a plams setting object.

qmctorch.scf.calculator.calculator_base module

class qmctorch.scf.calculator.calculator_base.**CalculatorBase**(*atoms*, *atom_coords*, *basis*, *scf*, *units*, *molname*, *calcname*, *savefile*)
Bases: object

run()

save_data()

static normalize_columns(*mat*)
Normalize a matrix column-wise.

Parameters **{np.ndarray} -- the matrix to be normalized**(*mat*) –

Returns np.ndarray – normalized matrix

qmctorch.scf.calculator.pyscf module

class qmctorch.scf.calculator.pyscf.**CalculatorPySCF**(*atoms*, *atom_coords*, *basis*, *scf*, *units*, *molname*, *savefile*)
Bases: *qmctorch.scf.calculator.calculator_base.CalculatorBase*

run()
Run the scf calculation using PySCF.

get_basis_data(*mol*, *rhf*)
Save the data to HDF5

Parameters

- **{pyscf.gto.M} -- psycf Molecule** (*mol*) –
- **{pyscf.scf} -- scf object** (*rhf*) –

get_atoms_str()
Refresh the atom string (use after atom move).

static get_bas_n(*mol*)

Module contents

class qmctorch.scf.calculator.**CalculatorBase**(*atoms*, *atom_coords*, *basis*, *scf*, *units*, *molname*, *calcname*, *savefile*)

Bases: object

run()

save_data()

static normalize_columns(*mat*)
Normalize a matrix column-wise.

Parameters **{np.ndarray} -- the matrix to be normalized** (*mat*) –

Returns np.ndarray – normalized matrix

class qmctorch.scf.calculator.**CalculatorADF**(*atoms*, *atom_coords*, *basis*, *scf*, *units*, *molname*, *savefile*)

Bases: *qmctorch.scf.calculator.calculator_base.CalculatorBase*

run()
Run the calculation using ADF.

init_plams()
Init PLAMS.

get_plams_molecule()
Returns a plams molecule object.

get_plams_settings()
Returns a plams setting object.

get_basis_data(*kffile*)
Save the basis information needed to compute the AO values.

static read_array(*kf*, *section*, *name*)
read a data from the kf file

Parameters

- **kf** (*file handle*) – kf file
- **section** (*str*) – name of the section
- **name** (*str*) – name of the property

Returns data

Return type np.data

class qmctorch.scf.calculator.**CalculatorADF2019**(*atoms*, *atom_coords*, *basis*, *scf*, *units*, *molname*, *savefile*)

Bases: *qmctorch.scf.calculator.adf.CalculatorADF*

get_plams_molecule()
Returns a plams molecule object.

get_plams_settings()
Returns a plams setting object.

class qmctorch.scf.calculator.**CalculatorPySCF**(*atoms*, *atom_coords*, *basis*, *scf*, *units*, *molname*, *savefile*)

Bases: *qmctorch.scf.calculator.calculator_base.CalculatorBase*

run()
Run the scf calculation using PySCF.

get_basis_data(*mol*, *rhf*)
Save the data to HDF5

Parameters

- **{pyscf.gto.M} -- psycf Molecule** (*mol*) –
- **{pyscf.scf} -- scf object** (*rhf*) –

get_atoms_str()
Refresh the atom string (use after atom move).

static get_bas_n(*mol*)

14.2.2 Submodules

qmctorch.scf.molecule module

class qmctorch.scf.molecule.**Molecule**(*atom=None*, *calculator='pyscf'*, *scf='hf'*, *basis='sto-3g'*, *unit='bohr'*, *name=None*, *load=None*, *save_scf_file=False*, *redo_scf=False*, *rank=0*)

Bases: object

Create a molecule in QMCTorch

Parameters

- **atom** (*str or None, optional*) – defines the atoms and their positions. Defaults to None. - At1 x y z; At2 x y z ... : Provide the atomic coordinate directly - <file>.xyz : provide the path to an .xyz file containing the atomic coordinates
- **calculator** (*str, optional*) – selet scf calculator. Defaults to 'adf'. - pyscf : PySCF calculator - adf : ADF2020+ calculator - adf2019 : ADF2019 calculatori
- **scf** (*str, optional*) – select scf level of theory. Defaults to 'hf'. - hf : perform a Hatree-Fock calculation to obtain the molecular orbital coefficients - dft : perform a density functional theory using the local density approximation
- **basis** (*str, optional*) – select the basis set. Defaults to 'dzp'.
- **unit** (*str, optional*) – units of the coordinates; 'bohr' or 'angs'. Defaults to 'bohr'.
- **name** (*str or None, optional*) – name of the molecule. Defaults to None.
- **load** (*str or None, optional*) – path to a hdf5 file to load. Defaults to None.

- **save_scf_file** (*bool, optional*) – save the scf file (when applicable) Defaults to False
- **redo_scf** (*bool, optional*) – if true ignore existing hdf5 file and redo the scf calculation
- **rank** (*int, optional*) – Rank of the process. Defaults to 0.

Examples

```
>>> from qmctorch.scf import Molecule
>>> mol = Molecule(atom='H 0 0 0; H 0 0 1', unit='angs',
...                calculator='adf', basis='dzp')
```

log_data()

domain(*method*)

Returns information to initialize the walkers

Parameters **method** (*str*) – 'center', all electron at the center of the system 'uniform', all electrons in a cube surrounding the molecule 'normal', all electrons in a sphere surrounding the molecule 'atomic', electrons around the atoms

Returns dictionary containing corresponding information

Return type dict

Examples::

```
>>> mol = Molecule('h2.xyz')
>>> domain = mol.domain('atomic')
```

print_total_energy()

Print the SCF energy of the molecule.

Examples::

```
>>> mol = Molecule('h2.xyz', calculator='adf', basis='sz')
>>> mol.print_total_energy()
```

get_total_energy()

Get the value of the total energy.

14.2.3 Module contents

class qmctorch.scf.**Molecule**(*atom=None, calculator='pyscf', scf='hf', basis='sto-3g', unit='bohr', name=None, load=None, save_scf_file=False, redo_scf=False, rank=0*)

Bases: object

Create a molecule in QMCTorch

Parameters

- **atom** (*str or None, optional*) – defines the atoms and their positions. Defaults to None. - At1 x y z; At2 x y z ... : Provide the atomic coordinate directly - <file>.xyz : provide the path to an .xyz file containing the atomic coordinates

- **calculator** (*str, optional*) – selet scf calculator. Defaults to 'adf'. - pyscf : PySCF calculator - adf : ADF2020+ calculator - adf2019 : ADF2019 calculatori
- **scf** (*str, optional*) – select scf level of theory. Defaults to 'hf'. - hf : perform a Hatree-Fock calculation to obtain the molecular orbital coefficients - dft : perform a density functional theory using the local density approximation
- **basis** (*str, optional*) – select the basis set. Defaults to 'dzp'.
- **unit** (*str, optional*) – units of the coordinates; 'bohr' or 'angs'. Defaults to 'bohr'.
- **name** (*str or None, optional*) – name of the molecule. Defaults to None.
- **load** (*str or None, optional*) – path to a hdf5 file to load. Defaults to None.
- **save_scf_file** (*bool, optional*) – save the scf file (when applicable) Defaults to False
- **redo_scf** (*bool, optional*) – if true ignore existing hdf5 file and redo the scf calculation
- **rank** (*int, optional*) – Rank of the process. Defaults to 0.

Examples

```
>>> from qmctorch.scf import Molecule
>>> mol = Molecule(atom='H 0 0 0; H 0 0 1', unit='angs',
...                calculator='adf', basis='dzp')
```

log_data()

domain(*method*)

Returns information to initialize the walkers

Parameters **method** (*str*) – 'center', all electron at the center of the system 'uniform', all electrons in a cube surrounding the molecule 'normal', all electrons in a sphere surrounding the molecule 'atomic', electrons around the atoms

Returns dictionary containing corresponding information

Return type dict

Examples::

```
>>> mol = Molecule('h2.xyz')
>>> domain = mol.domain('atomic')
```

print_total_energy()

Print the SCF energy of the molecule.

Examples::

```
>>> mol = Molecule('h2.xyz', calculator='adf', basis='sz')
>>> mol.print_total_energy()
```

get_total_energy()

Get the value of the total energy.

14.3 qmctorch.solver package

14.3.1 Submodules

qmctorch.solver.solver_base module

class qmctorch.solver.solver_base.**SolverBase**(*wf=None*, *sampler=None*, *optimizer=None*, *scheduler=None*, *output=None*, *rank=0*)

Bases: object

Base Class for QMC solver

Parameters

- **wf** (*qmctorch.WaveFunction, optional*) – wave function. Defaults to None.
- **sampler** (*qmctorch.sampler, optional*) – Sampler. Defaults to None.
- **optimizer** (*torch.optim, optional*) – optimizer. Defaults to None.
- **scheduler** (*torch.optim, optional*) – scheduler. Defaults to None.
- **output** (*str, optional*) – hdf5 filename. Defaults to None.
- **rank** (*int, optional*) – rank of he process. Defaults to 0.

configure_resampling(*mode='update'*, *resample_every=1*, *nstep_update=25*)
Configure the resampling

Parameters

- **mode** (*str, optional*) – method to resample : 'full', 'update', 'never' Defaults to 'update'.
- **resample_every** (*int, optional*) – Number of optimization steps between resampling Defaults to 1.
- **nstep_update** (*int, optional*) – Number of MC steps in update mode. Defaults to 25.

track_observable(*obs_name*)
define the observalbe we want to track

Parameters **obs_name** (*list*) – list of str defining the observalbe. Each str must correspond to a WaveFuncion method

store_observable(*pos*, *local_energy=None*, *ibatch=None*, ***kwargs*)
store observale in the dictionary

Parameters

- **obs_dict** (*dict*) – dictionary of the observalbe
- **pos** (*torch.tensor*) – positions of th walkers
- **local_energy** (*torch.tensor, optional*) – precomputed values of the local energy. Defaults to None
- **ibatch** (*int*) – index of the current batch. Defaults to None

print_observable(*cumulative_loss*, *verbose=False*)
Print the observalbe to csreen

Parameters

- **cumulative_loss** (*float*) – current loss value
- **verbose** (*bool, optional*) – print all the observables. Defaults to False

resample(*n*, *pos*)
Resample the wave function

Parameters

- **n** (*int*) – current epoch value
- **pos** (*torch.tensor*) – positions of the walkers

Returns new positions of the walkers

Return type (torch.tensor)

single_point(*with_tqdm=True*, *hdf5_group='single_point'*)
Performs a single point calculatin

Parameters

- **with_tqdm** (*bool, optional*) – use tqdm for samplig. Defaults to True.
- **hdf5_group** (*str, optional*) – hdf5 group where to store the data. Defaults to 'single_point'.

Returns contains the local energy, positions, ...

Return type SimpleNamespace

save_checkpoint(*epoch*, *loss*)
save the model and optimizer state

Parameters

- **epoch** (*int*) – epoch
- **loss** (*float*) – current value of the loss
- **filename** (*str*) – name to save the file

load_checkpoint(*filename*)
load a model/optmizer

Parameters **filename** (*str*) – filename

Returns epoch number and loss

Return type tuple

sampling_traj(*pos=None*, *with_tqdm=True*, *hdf5_group='sampling_trajectory'*)
Compute the local energy along a sampling trajectory

Parameters

- **pos** (*torch.tensor*) – positions of the walkers along the trajectory
- **hdf5_group** (*str, optional*) – name of the group where to store the data. Defaults to 'sampling_trajecory'

Returns contains energy/positions/

Return type SimpleNamespace

print_parameters(*grad=False*)
print parameter values

Parameters **grad** (*bool, optional*) – also print the gradient. Defaults to False.

optimization_step(*lpos*)
Performs one optimization step

Parameters **{torch.tensor} -- positions of the walkers**(*lpos*) –

save_traj(*fname*, *obs*)
Save trajectory of geo_opt

Parameters **fname**(*str*) – file name

run(*nepoch*, *batchsize=None*, *loss='variance'*)

log_data()
Log basic information about the sampler.

14.3.2 Module contents

class qmctorch.solver.**SolverBase**(*wf=None*, *sampler=None*, *optimizer=None*, *scheduler=None*, *output=None*, *rank=0*)

Bases: object

Base Class for QMC solver

Parameters

- **wf**(*qmctorch.WaveFunction, optional*) – wave function. Defaults to None.
- **sampler**(*qmctorch.sampler, optional*) – Sampler. Defaults to None.
- **optimizer**(*torch.optim, optional*) – optimizer. Defaults to None.
- **scheduler**(*torch.optim, optional*) – scheduler. Defaults to None.
- **output**(*str, optional*) – hdf5 filename. Defaults to None.
- **rank**(*int, optional*) – rank of he process. Defaults to 0.

configure_resampling(*mode='update'*, *resample_every=1*, *nstep_update=25*)
Configure the resampling

Parameters

- **mode**(*str, optional*) – method to resample : 'full', 'update', 'never' Defaults to 'update'.
- **resample_every**(*int, optional*) – Number of optimization steps between resampling Defaults to 1.
- **nstep_update**(*int, optional*) – Number of MC steps in update mode. Defaults to 25.

track_observable(*obs_name*)
define the observalbe we want to track

Parameters **obs_name**(*list*) – list of str defining the observalbe. Each str must correspond to a WaveFuncion method

store_observable(*pos*, *local_energy=None*, *ibatch=None*, ***kwargs*)
store observale in the dictionary

Parameters

- **obs_dict**(*dict*) – dictionary of the observalbe
- **pos**(*torch.tensor*) – positions of th walkers

- **local_energy** (*torch.tensor, optional*) – precomputed values of the local energy. Defaults to None
- **ibatch** (*int*) – index of the current batch. Defaults to None

print_observable (*cumulative_loss*, *verbose=False*)
Print the observalbe to csreen

Parameters

- **cumulative_loss** (*float*) – current loss value
- **verbose** (*bool, optional*) – print all the observables. Defaults to False

resample (*n*, *pos*)
Resample the wave function

Parameters

- **n** (*int*) – current epoch value
- **pos** (*torch.tensor*) – positions of the walkers

Returns new positions of the walkers

Return type (torch.tensor)

single_point (*with_tqdm=True*, *hdf5_group='single_point'*)
Performs a single point calculatin

Parameters

- **with_tqdm** (*bool, optional*) – use tqdm for samplig. Defaults to True.
- **hdf5_group** (*str, optional*) – hdf5 group where to store the data. Defaults to 'single_point'.

Returns contains the local energy, positions, ...

Return type SimpleNamespace

save_checkpoint (*epoch*, *loss*)
save the model and optimizer state

Parameters

- **epoch** (*int*) – epoch
- **loss** (*float*) – current value of the loss
- **filename** (*str*) – name to save the file

load_checkpoint (*filename*)
load a model/optmizer

Parameters **filename** (*str*) – filename

Returns epoch number and loss

Return type tuple

sampling_traj (*pos=None*, *with_tqdm=True*, *hdf5_group='sampling_trajectory'*)
Compute the local energy along a sampling trajectory

Parameters

- **pos** (*torch.tensor*) – positions of the walkers along the trajectory

- **hdf5_group** (*str, optional*) – name of the group where to store the data. Defaults to 'sampling_trajecory'

Returns contains energy/positions/

Return type SimpleNamespace

print_parameters(*grad=False*)
print parameter values

Parameters **grad** (*bool, optional*) – also print the gradient. Defaults to False.

optimization_step(*lpos*)
Performs one optimization step

Parameters **{torch.tensor} -- positions of the walkers** (*lpos*) –

save_traj(*fname*, *obs*)
Save trajectory of geo_opt

Parameters **fname** (*str*) – file name

run(*nepoch*, *batchsize=None*, *loss='variance'*)

log_data()
Log basic information about the sampler.

class qmctorch.solver.**Solver**(*wf=None*, *sampler=None*, *optimizer=None*, *scheduler=None*, *output=None*, *rank=0*)
Bases: *qmctorch.solver.solver_base.SolverBase*

Basic QMC solver

Parameters

- **wf** (*qmctorch.WaveFunction, optional*) – wave function. Defaults to None.
- **sampler** (*qmctorch.sampler, optional*) – Sampler. Defaults to None.
- **optimizer** (*torch.optim, optional*) – optimizer. Defaults to None.
- **scheduler** (*torch.optim, optional*) – scheduler. Defaults to None.
- **output** (*str, optional*) – hdf5 filename. Defaults to None.
- **rank** (*int, optional*) – rank of he process. Defaults to 0.

configure(*track=None*, *freeze=None*, *loss=None*, *grad=None*, *ortho_mo=None*, *clip_loss=False*, *resampling=None*)
Configure the solver

Parameters

- **track** (*list, optional*) – list of observable to track. Defaults to ['local_energy'].
- **freeze** (*[type], optional*) – list of parameters to freeze. Defaults to None.
- **loss** (*str, optional*) – merhod to compute the loss: variance or energy. Defaults to 'energy'.
- **grad** (*str, optional*) – method to compute the gradients: 'auto' or 'manual'. Defaults to 'auto'.
- **ortho_mo** (*bool, optional*) – apply regularization to orthogonalize the MOs. Defaults to False.
- **clip_loss** (*bool, optional*) – Clip the loss values at +/- X std. X defined in Loss as clip_num_std (default 5) Defaults to False.

set_params_requires_grad(*wf_params=True*, *geo_params=False*)
Configure parameters for wf opt.

freeze_parameters(*freeze*)
Freeze the optimization of specified params.

Parameters **freeze** (*list*) – list of param to freeze

save_sampling_parameters(*pos*)
save the sampling params.

restore_sampling_parameters()
restore sampling params to their original values.

geo_opt(*nepoch*, *geo_lr=0.01*, *batchsize=None*, *nepoch_wf_init=100*, *nepoch_wf_update=50*, *hdf5_group='geo_opt'*, *chkpt_every=None*, *tqdm=False*)
optimize the geometry of the molecule

Parameters

- **nepoch** (*int*) – Number of optimziation step
- **batchsize** (*int, optional*) – Number of sample in a mini batch. If None, all samples are used. Defaults to Never.
- **hdf5_group** (*str, optional*) – name of the hdf5 group where to store the data. Defaults to 'geo_opt'.
- **chkpt_every** (*int, optional*) – save a checkpoint every every iteration. Defaults to half the number of epoch

run(*nepoch*, *batchsize=None*, *hdf5_group='wf_opt'*, *chkpt_every=None*, *tqdm=False*)
Run a wave function optimization

Parameters

- **nepoch** (*int*) – Number of optimziation step
- **batchsize** (*int, optional*) – Number of sample in a mini batch. If None, all samples are used. Defaults to Never.
- **hdf5_group** (*str, optional*) – name of the hdf5 group where to store the data. Defaults to 'wf_opt'.
- **chkpt_every** (*int, optional*) – save a checkpoint every every iteration. Defaults to half the number of epoch

prepare_optimization(*batchsize*, *chkpt_every*, *tqdm=False*)
Prepare the optimization process

Parameters

- **batchsize** (*int or None*) – batchsize
- **chkpt_every** (*int or none*) – save a chkpt file every

save_data(*hdf5_group*)
Save the data to hdf5.

Parameters **hdf5_group** (*str*) – name of group in the hdf5 file

run_epochs(*nepoch*)
Run a certain number of epochs

Parameters **nepoch** (*int*) – number of epoch to run

evaluate_grad_auto(*lpos*)
Evaluate the gradient using automatic differentiation

Parameters **lpos** (*torch.tensor*) – sampling points

Returns loss values and local energies

Return type tuple

evaluate_grad_manual(*lpos*)
Evaluate the gradient using low variance expression

Parameters

- **lpos** (*torch.tensor*) – [description]
- **lpos** – sampling points

Returns loss values and local energies

Return type tuple

log_data_opt(*nepoch*, *task*)
Log data for the optimization.

class qmctorch.solver.**SolverMPI**(*wf=None*, *sampler=None*, *optimizer=None*, *scheduler=None*, *output=None*, *rank=0*)
Bases: qmctorch.solver.solver.Solver

Distributed QMC solver

Parameters

- **wf** (*qmctorch.WaveFunction, optional*) – wave function. Defaults to None.
- **sampler** (*qmctorch.sampler, optional*) – Sampler. Defaults to None.
- **optimizer** (*torch.optim, optional*) – optimizer. Defaults to None.
- **scheduler** (*torch.optim, optional*) – scheduler. Defaults to None.
- **output** (*str, optional*) – hdf5 filename. Defaults to None.
- **rank** (*int, optional*) – rank of he process. Defaults to 0.

run(*nepoch*, *batchsize=None*, *loss='energy'*, *clip_loss=False*, *grad='manual'*, *hdf5_group='wf_opt'*, *num_threads=1*, *chkpt_every=None*)
Run the optimization

Parameters

- **nepoch** (*int*) – Number of optimization step
- **batchsize** (*int, optional*) – Number of sample in a mini batch. If None, all samples are used. Defaults to None.
- **loss** (*str, optional*) – method to compute the loss: variance or energy. Defaults to 'energy'.
- **clip_loss** (*bool, optional*) – Clip the loss values at +/- 5std. Defaults to False.
- **grad** (*str, optional*) – method to compute the gradients: 'auto' or 'manual'. Defaults to 'auto'.
- **hdf5_group** (*str, optional*) – name of the hdf5 group where to store the data. Defaults to 'wf_opt'

single_point(*with_tqdm=True*, *hdf5_group='single_point'*)
Performs a single point calculation

Parameters

- **with_tqdm** (*bool, optional*) – use tqdm for samplig. Defaults to True.
- **hdf5_group** (*str, optional*) – hdf5 group where to store the data. Defaults to 'single_point'.

Returns contains the local energy, positions, ...

Return type SimpleNamespace

static metric_average(*val*, *name*)
Average a give quantity over all processes

Parameters

- **{torch.tensor} -- data to average** (*val*) –
- **{str} -- name of the data** (*name*) –

Returns torch.tensor – Averaged quantity

14.4 qmctorch.utils package

14.4.1 Submodules

qmctorch.utils.algebra_utils module

qmctorch.utils.algebra_utils.**btrace**(*M*)
Computes the trace of batched matrices

Parameters **M** (*torch.tensor*) – matrices of size (Na, Nb, ... Nx, N, N)

Returns trace of matrices (Na, Nb, ... Nx)

Return type torch.tensor

Example

```
>>> m = torch.rand(100,5,5)
>>> tr = btrace(m)
```

qmctorch.utils.algebra_utils.**bproj**(*M*, *P*)
Project batched marices using P^T M P

Parameters

- **M** (*torch.tensor*) – batched matrices size (..., N, M)
- **P** (*torch.tensor*) – Porjectors size (..., N, M)

Returns Projected matrices

Return type torch.tensor

qmctorch.utils.algebra_utils.**bdet2**(*M*)
Computes the determinant of batched 2x2 matrices

Parameters **M** (*torch.tensor*) – input matrices

Returns determinants of the matrices

Return type torch.tensor

class qmctorch.utils.algebra_utils.**BatchDeterminant**(**args*, ***kwargs*)
Bases: sphinx.ext.autodoc.importer._MockObject

static forward(*ctx*, *input*)

static backward(*ctx*, *grad_output*)

using jaobi's formula d det(A) / d A_{ij} = adj^T(A)_{ij}

using the adjunct formula d det(A) / d A_{ij} = ((det(A) A^{-1})^T)_{ij}

qmctorch.utils.hdf5_utils module

qmctorch.utils.hdf5_utils.**print_insert_error**(*obj*, *obj_name*)

qmctorch.utils.hdf5_utils.**print_insert_type_error**(*obj*, *obj_name*)

qmctorch.utils.hdf5_utils.**print_load_error**(*grp*)

qmctorch.utils.hdf5_utils.**load_from_hdf5**(*obj*, *fname*, *obj_name*)
Load the content of an hdf5 file in an object.

Parameters

- **{object} -- object where to load the data** (*obj*) –
- **{str} -- name pf the hdf5 file** (*fname*) –
- **{str} -- name of the root group in the hdf5** (*obj_name*) –

qmctorch.utils.hdf5_utils.**load_object**(*grp*, *parent_obj*, *grp_name*)
Load object attribute from the hdf5 group/data

Parameters

- **{hdf5 group} -- the current group in the hdf5 architecture** (*grp*) –
- **{object} -- parent object** (*parent_obj*) –
- **{str} -- name of the group** (*grp_name*) –

qmctorch.utils.hdf5_utils.**load_group**(*grp*, *parent_obj*, *grp_name*)
Load object attribute from the hdf5 group

Parameters

- **{hdf5 group} -- the current group in the hdf5 architecture** (*grp*) –
- **{object} -- parent object** (*parent_obj*) –
- **{str} -- name of the group** (*grp_name*) –

qmctorch.utils.hdf5_utils.**load_data**(*grp*, *parent_obj*, *grp_name*)
Load data from the hdf5 data

Parameters

- **{hdf5 group} -- the current group in the hdf5 architecture** (*grp*) –

- **{object} -- parent object** (*parent_obj*) –
- **{str} -- name of the group** (*grp_name*) –

qmctorch.utils.hdf5_utils.**cast_loaded_data**(*data*)
cast the data before loading.

qmctorch.utils.hdf5_utils.**bytes2str**(*bstr*)
Convert a bytes into string.

qmctorch.utils.hdf5_utils.**lookup_cast**(*ori_type*, *current_type*)

qmctorch.utils.hdf5_utils.**isgroup**(*grp*)
Check if current hdf5 group is a group

Parameters **{hdf5 group} -- hdf5 group or dataset** (*grp*) –

Returns bool – True if the group is a group

qmctorch.utils.hdf5_utils.**dump_to_hdf5**(*obj*, *fname*, *root_name=None*)
Dump the content of an object in a hdf5 file.

Parameters

- **{object} -- object to dump** (*obj*) –
- **{str} -- name of the hdf5** (*fname*) –

Keyword Arguments **{str} -- root group in the hdf5 file (default** (*root_name*) – {None})

qmctorch.utils.hdf5_utils.**insert_object**(*obj*, *parent_grp*, *obj_name*)
Insert the content of the object in the hdf5 file

Parameters

- **{object} -- object to save** (*obj*) –
- **{hdf5 group} -- group where to dump** (*parent_grp*) –
- **{str} -- name of the object** (*obj_name*) –

qmctorch.utils.hdf5_utils.**insert_group**(*obj*, *parent_grp*, *obj_name*)
Insert the content of the object in a hdf5 group

Parameters

- **{object} -- object to save** (*obj*) –
- **{hdf5 group} -- group where to dump** (*parent_grp*) –
- **{str} -- name of the object** (*obj_name*) –

qmctorch.utils.hdf5_utils.**insert_data**(*obj*, *parent_grp*, *obj_name*)
Insert the content of the object in a hdf5 dataset

Parameters

- **{object} -- object to save** (*obj*) –
- **{hdf5 group} -- group where to dump** (*parent_grp*) –
- **{str} -- name of the object** (*obj_name*) –

qmctorch.utils.hdf5_utils.**insert_type**(*obj*, *parent_grp*, *obj_name*)
Insert the content of the type object in an attribute

Parameters

- **{object} -- object to save**(*obj*) –
- **{hdf5 group} -- group where to dump**(*parent_grp*) –
- **{str} -- name of the object**(*obj_name*) –

qmctorch.utils.hdf5_utils.**insert_default**(*obj*, *parent_grp*, *obj_name*)
Default funtion to insert a dataset

Parameters

- **{object} -- object to save**(*obj*) –
- **{hdf5 group} -- group where to dump**(*parent_grp*) –
- **{str} -- name of the object**(*obj_name*) –

qmctorch.utils.hdf5_utils.**insert_list**(*obj*, *parent_grp*, *obj_name*)
funtion to insert a list as a dataset

Parameters

- **{object} -- object to save**(*obj*) –
- **{hdf5 group} -- group where to dump**(*parent_grp*) –
- **{str} -- name of the object**(*obj_name*) –

qmctorch.utils.hdf5_utils.**insert_tuple**(*obj*, *parent_grp*, *obj_name*)
funtion to insert a tuple as a dataset

Parameters

- **{object} -- object to save**(*obj*) –
- **{hdf5 group} -- group where to dump**(*parent_grp*) –
- **{str} -- name of the object**(*obj_name*) –

qmctorch.utils.hdf5_utils.**insert_numpy**(*obj*, *parent_grp*, *obj_name*)
funtion to insert a numpy array as a dataset

Parameters

- **{object} -- object to save**(*obj*) –
- **{hdf5 group} -- group where to dump**(*parent_grp*) –
- **{str} -- name of the object**(*obj_name*) –

qmctorch.utils.hdf5_utils.**insert_torch_tensor**(*obj*, *parent_grp*, *obj_name*)
funtion to insert a torch tensor as a dataset

Parameters

- **{object} -- object to save**(*obj*) –
- **{hdf5 group} -- group where to dump**(*parent_grp*) –
- **{str} -- name of the object**(*obj_name*) –

qmctorch.utils.hdf5_utils.**insert_torch_parameter**(*obj*, *parent_grp*, *obj_name*)
funtion to insert a torch parameter as a dataset

Parameters

- **{object} -- object to save**(*obj*) –
- **{hdf5 group} -- group where to dump**(*parent_grp*) –

- **{str} -- name of the object** (*obj_name*) –

qmctorch.utils.hdf5_utils.**insert_none**(*obj*, *parent_grp*, *obj_name*)
funtion to insert a None Type as a dataset

Parameters

- **{object} -- object to save** (*obj*) –
- **{hdf5 group} -- group where to dump** (*parent_grp*) –
- **{str} -- name of the object** (*obj_name*) –

qmctorch.utils.hdf5_utils.**haschildren**(*obj*)
Check if the object has children

Parameters **{object} -- the object to check** (*obj*) –

Returns bool – True if the object has children

qmctorch.utils.hdf5_utils.**children**(*obj*)
Returns the children of the object as items

Parameters **{object} -- the object to check** (*obj*) –

Returns dict – items

qmctorch.utils.hdf5_utils.**get_children_names**(*obj*)
Returns the children names of the object as items

Parameters **{object} -- the object to check** (*obj*) –

Returns dict – items

qmctorch.utils.hdf5_utils.**get_child_object**(*obj*, *child_name*)
Return the child object

Parameters

- **{object} -- parent object** (*obj*) –
- **{str} -- cild name** (*child_name*) –

Returns object – child object

qmctorch.utils.hdf5_utils.**add_group_attr**(*filename*, *grp_name*, *attr*)
Add attribute to a given group

Parameters

- **{str} -- name of the file** (*filename*) –
- **{str} -- name of the group** (*grp_name*) –
- **{dict} -- attrivutes to add** (*attr*) –

qmctorch.utils.hdf5_utils.**register_extra_attributes**(*obj*, *attr_names*)
Register extra attribute to be able to dump them

Parameters

- **{object} -- the object where we want to add attr** (*obj*) –
- **{list} -- a list of attr names** (*attr_names*) –

qmctorch.utils.interpolate module

class qmctorch.utils.interpolate.**InterpolateMolecularOrbitals**(*wf*)
Bases: object

Interpolation of the AO using a log grid centered on each atom.

get_mo_max_index(*orb*)
Get the index of the highest MO to inlcude in the interpoaltion

Parameters **orb** (*str*) – occupied or all

Raises ValueError – if orb not valid

interpolate_mo_irreg_grid(*pos*, *n*, *orb*)
Interpolate the mo occupied in the configs.

Parameters

- **pos** (*torch.tensor*) – sampling points (Nbatch, 3*Nelec)
- **n** (*int, optional*) – Interpolation order. Defaults to 6.

Returns mo values Nbatch, Nelec, Nmo

Return type torch.tensor

interpolate_mo_reg_grid(*pos*, *res*, *blength*, *orb*)
Interpolate the mo occupied in the configs.

Parameters **pos** (*torch.tensor*) – sampling points (Nbatch, 3*Nelec)

Returns mo values Nbatch, Nelec, Nmo

Return type torch.tensor

class qmctorch.utils.interpolate.**InterpolateAtomicOrbitals**(*wf*)
Bases: object

Interpolation of the AO using a log grid centered on each atom.

get_interpolator(*n=6*, *length=2*)
evaluate the interpolation function.

Parameters

- **n** (*int, optional*) – number of points on each log axis. Defaults to 6.
- **length** (*int, optional*) – half length of the grid. Defaults to 2.

qmctorch.utils.interpolate.**get_boundaries**(*atomic_positions*, *border_length=2.0*)
Computes the boundaries of the structure

Parameters

- **atomic_positions** (*torch.Tensor, np.ndarray, list*) – atomic positions
- **border_length** (*float, optional*) – length of the border. Defaults to 2.

Raises ValueError – if type of positions not recognized

Returns min, max values in the 3 cartesian directions

Return type (np.ndarray, np.ndarray, mp.ndarray)

qmctorch.utils.interpolate.**get_reg_grid**(*atomic_positions*, *resolution=0.1*, *border_length=2.0*)
Computes a regular grid points from the atomic positions

Parameters

- **atomic_positions** (*torch.Tensor, np.ndarray, list*) – atomic positions
- **resolution** (*float, optional*) – ditance between two points. Defaults to 0.5.
- **border_length** (*float, optional*) – length of the border. Defaults to 2.

Returns grid points in the x, y and z axis

Return type (np.ndarray, np.ndarray, mp.ndarray)

qmctorch.utils.interpolate.**interpolator_reg_grid**(*func*, *x*, *y*, *z*)
Computes the interpolation function

Parameters

- **func** (*callable*) – compute the value of the funtion to interpolate
- **x** (*np.ndarray*) – grid points in the x direction
- **y** (*np.ndarray*) – grid points in the y direction
- **z** (*np.ndarray*) – grid points in the z direction

Returns interpolation function

Return type callable

qmctorch.utils.interpolate.**interpolate_reg_grid**(*interpfunc*, *pos*)
Interpolate the funtion

Parameters

- **interpfunc** (*callable*) – function to interpolate the data points
- **pos** (*torch.tensor*) – positions of the walkers Nbatch x 3*Nelec

Returns interpolated values of the function evaluated at pos

Return type torch.tensor

qmctorch.utils.interpolate.**is_even**(*x*)
return true if x is even.

qmctorch.utils.interpolate.**logspace**(*n*, *length*)
returns a 1d array of logspace between -length and +length.

qmctorch.utils.interpolate.**get_log_grid**(*atomic_positions*, *n=6*, *length=2.0*, *border_length=2.0*)
Computes a logarithmic grid

Parameters

- **atomic_positions** (*list, np.ndarray, torch.tensor*) – positions of the atoms
- **n** (*int, optional*) – number of points in each axis around each atom. Defaults to 6.
- **length** (*float, optional*) – absolute value of the max distance from the atom. Defaults to 2.
- **border_length** (*float, optional*) – length of the border. Defaults to 2.

Returns grid points (Npts,3)

Return type np.ndanrray

qmctorch.utils.interpolate.**interpolator_irreg_grid**(*func*, *grid_pts*)
compute a linear ND interpolator

Parameters

- **func** (*callable*) – compute the value of the funtion to interpolate
- **grid_pts** (*np.ndarray*) – grid points in the x direction

Returns interpolation function

Return type callable

qmctorch.utils.interpolate.**interpolate_irreg_grid**(*interpfunc*, *pos*)
Interpolate the funtion

Parameters

- **interpfunc** (*callable*) – function to interpolate the data points
- **pos** (*torch.tensor*) – positions of the walkers Nbatch x 3*Nelec

Returns interpolated values of the function evaluated at pos

Return type torch.tensor

qmctorch.utils.plot_data module

qmctorch.utils.plot_data.**plot_energy**(*local_energy*, *e0=None*, *show_variance=False*)
Plot the evolution of the energy

Parameters

- **local_energy** (*np.ndarray*) – local energies along the trajectory
- **e0** (*float, optional*) – Target value for the energy. Defaults to None.
- **show_variance** (*bool, optional*) – show the variance if True. Defaults to False.

qmctorch.utils.plot_data.**plot_data**(*observable*, *obsname*)
Plot the evolution a given data

Parameters

- **obs_dict** (*SimpleNamespace*) – namespace of observable
- **obsname** (*str*) – name (key) of the desired observable

qmctorch.utils.plot_data.**plot_walkers_traj**(*eloc*, *walkers='mean'*)
Plot the trajectory of all the individual walkers

Parameters

- **obs** (*SimpleNamespace*) – Namespace of the observables
- **walkers** (*int, str, optional*) – all, mean or index of a given walker Defaults to 'all'

qmctorch.utils.plot_data.**plot_correlation_coefficient**(*eloc*, *size_max=100*)

Plot the correlation coefficient of the local energy and fit the curve to an exp to extract the correlation time.

Parameters

- **eloc** (*np.ndarray*) – values of the local energy (Nstep, Nwalk)
- **size_max** (*int, optional*) – maximu number of MC step to consider.Defaults to 100.

Returns correlation coefficients (size_max, Nwalkers), correlation time

Return type np.ndarray, float

qmctorch.utils.plot_data.**plot_integrated_autocorrelation_time**(*eloc*, *rho=None*, *size_max=100*, *C=5*)

compute/plot the integrated autocorrelation time

Parameters

- **eloc** (*np.ndarray, optional*) – local energy values (Nstep, Nwalkers)
- **rho** (*np.ndarray, optional*) – Correlation coefficient. Defaults to None.
- **size_max** (*int, optional*) – maximu number of MC step to consider.Defaults to 100.
- **C** (*int, optional*) – [description]. Defaults to 5.

qmctorch.utils.plot_data.**plot_blocking_energy**(*eloc*, *block_size*, *walkers='mean'*)

Plot the blocked energy values

Parameters

- **eloc** (*np.ndarray*) – values of the local energies
- **block_size** (*int*) – size of the block
- **walkers** (*str, optional*) – which walkers to plot (mean, all, index or list). Defaults to 'mean'.

Raises ValueError – [description]

qmctorch.utils.plot_data.**plot_correlation_time**(*eloc*)

Plot the blocking thingy

Parameters **eloc** (*np.array*) – values of the local energy

qmctorch.utils.plot_data.**plot_block**(*eloc*)

Plot the blocking thingy

Parameters **eloc** (*np.array*) – values of the local energy

qmctorch.utils.stat_utils module

qmctorch.utils.stat_utils.**blocking**(*x*, *block_size*, *expand=False*)

block the data

Parameters

- **x** (*data*) – size Nsample, Nexp
- **block_size** (*int*) – size of the block

qmctorch.utils.stat_utils.**correlation_coefficient**(*x*, *norm=True*)

Computes the correlation coefficient

Parameters

- **x** (*np.ndarray*) – measurement of size [Nsample, Nexperiments]
- **norm** (*bool, optional*) – [description]. Defaults to True.

qmctorch.utils.stat_utils.**integrated_autocorrelation_time**(*correlation_coeff*, *size_max*)

Computes the integrated autocorrelation time

Parameters

- **correlation_coeff** (*np.ndarray*) – coeff size Nsample,Nexp
- **size_max** (*int*) – max size

qmctorch.utils.stat_utils.**fit_correlation_coefficient**(*coeff*)

Fit the correlation coefficient to get the correlation time.

Parameters **coeff** (*np.ndarray*) – correlation coefficient

Returns correlation time, fitted curve

Return type float, np.ndarray

qmctorch.utils.torch_utils module

qmctorch.utils.torch_utils.**set_torch_double_precision**()
Set the default precision to double for all torch tensors.

qmctorch.utils.torch_utils.**set_torch_single_precision**()
Set the default precision to single for all torch tensors.

qmctorch.utils.torch_utils.**fast_power**(*x*, *k*, *mask0=None*, *mask2=None*)
Computes x**k when k have elements 0, 1, 2

Parameters

- **x** (*torch.tensor*) – input
- **k** (*torch.tensor*) – exponents
- **mask0** (*torch.tensor*) – precomputed mask of the elements of that are 0 (Defaults to None and computed here)
- **mask2** (*torch.tensor*) – precomputed mask of the elements of that are 2 (Defaults to None and computed here)

Returns values of x**k

Return type torch.tensor

qmctorch.utils.torch_utils.**gradients**(*out*, *inp*)
Return the gradients of out wrt inp

Parameters

- **out** (*[type]*) – [description]
- **inp** (*[type]*) – [description]

qmctorch.utils.torch_utils.**diagonal_hessian**(*out*, *inp*, *return_grads=False*)
return the diagonal hessian of out wrt to inp

Parameters

- **out** (*[type]*) – [description]
- **inp** (*[type]*) – [description]

Returns [description]

Return type [type]

class qmctorch.utils.torch_utils.**DataSet**(*data*)
Bases: sphinx.ext.autodoc.importer._MockObject

Creates a torch data set

Parameters **{torch.tensor} -- data** (*data*) –

class qmctorch.utils.torch_utils.**Loss**(*wf*, *method='energy'*, *clip=False*)
Bases: sphinx.ext.autodoc.importer._MockObject

Defines the loss to use during the optimization

Parameters **{WaveFunction} -- wave function object used** (*wf*) –

Keyword Arguments

- **{str} -- method to use (default** (*method*) – {'energy'}) (energy, variance, weighted-energy, weighted-variance)
- **{bool} -- clip the values that are +/- % sigma away from** (*clip*) – the mean (default: {False})

forward(*pos*, *no_grad=False*, *deactivate_weight=False*)
Computes the loss

Parameters **{torch.tensor} -- positions of the walkers in that batch** (*pos*) –

Keyword Arguments **{bool} -- computes the gradient of the loss** (*no_grad*) – (default: {False})

Returns torch.tensor, torch.tensor – value of the loss, local energies

static get_grad_mode(*no_grad*)
Returns enable_grad or no_grad

Parameters **{bool} -- [description]** (*no_grad*) –

get_clipping_mask(*local_energies*)
computes the clipping mask

Parameters **{torch.tensor} -- values of the local energies** (*local_energies*) –

get_sampling_weights(*pos*, *deactivate_weight*)
Get the weight needed when resampling is not done at every step

class qmctorch.utils.torch_utils.**OrthoReg**(*alpha=0.1*)
Bases: sphinx.ext.autodoc.importer._MockObject

Add a penalty loss to keep the MO orthogonalized

Keyword Arguments **{float} -- strength of the penaly (default** (*alpha*) – {0.1})

forward(*W*)
Return the loss : **|W x W^T - I|**.

14.4.2 Module contents

Utils module API.

qmctorch.utils.**plot_energy**(*local_energy*, *e0=None*, *show_variance=False*)
Plot the evolution of the energy

Parameters

- **local_energy** (*np.ndarray*) – local energies along the trajectory
- **e0** (*float, optional*) – Target value for the energy. Defaults to None.
- **show_variance** (*bool, optional*) – show the variance if True. Defaults to False.

qmctorch.utils.**plot_data**(*observable*, *obsname*)
Plot the evolution a given data

Parameters

- **obs_dict** (*SimpleNamespace*) – namespace of observable
- **obsname** (*str*) – name (key) of the desired observable

qmctorch.utils.**plot_block**(*eloc*)
Plot the blocking thingy

Parameters **eloc** (*np.array*) – values of the local energy

qmctorch.utils.**plot_walkers_traj**(*eloc*, *walkers='mean'*)
Plot the trajectory of all the individual walkers

Parameters

- **obs** (*SimpleNamespace*) – Namespace of the observables
- **walkers** (*int, str, optional*) – all, mean or index of a given walker Defaults to 'all'

qmctorch.utils.**plot_correlation_time**(*eloc*)
Plot the blocking thingy

Parameters **eloc** (*np.array*) – values of the local energy

qmctorch.utils.**set_torch_double_precision**()
Set the default precision to double for all torch tensors.

qmctorch.utils.**set_torch_single_precision**()
Set the default precision to single for all torch tensors.

class qmctorch.utils.**DataSet**(*data*)
Bases: sphinx.ext.autodoc.importer._MockObject

Creates a torch data set

Parameters **{torch.tensor} -- data** (*data*) –

class qmctorch.utils.**Loss**(*wf*, *method='energy'*, *clip=False*)
Bases: sphinx.ext.autodoc.importer._MockObject

Defines the loss to use during the optimization

Parameters **{WaveFunction} -- wave function object used** (*wf*) –

Keyword Arguments

- **{str} -- method to use (default** (*method*) – {'energy'}) (energy, variance, weighted-energy, weighted-variance)
- **{bool} -- clip the values that are +/- % sigma away from** (*clip*) – the mean (default: {False})

forward(*pos*, *no_grad=False*, *deactivate_weight=False*)
Computes the loss

Parameters **{torch.tensor} -- positions of the walkers in that batch** (*pos*) –

Keyword Arguments **{bool} -- computes the gradient of the loss** (*no_grad*) – (default: {False})

Returns torch.tensor, torch.tensor – value of the loss, local energies

static get_grad_mode(*no_grad*)
Returns enable_grad or no_grad

Parameters **{bool} -- [description]** (*no_grad*) –

get_clipping_mask(*local_energies*)
computes the clipping mask

Parameters **{torch.tensor} -- values of the local energies** (*local_energies*) –

get_sampling_weights(*pos*, *deactivate_weight*)
Get the weight needed when resampling is not done at every step

class qmctorch.utils.**OrthoReg**(*alpha=0.1*)
Bases: sphinx.ext.autodoc.importer._MockObject

Add a penalty loss to keep the MO orthogonalized

Keyword Arguments **{float} -- strength of the penaly (default** (*alpha*) – {0.1})

forward(*W*)
Return the loss : **|W x W^T - I|**.

qmctorch.utils.**dump_to_hdf5**(*obj*, *fname*, *root_name=None*)
Dump the content of an object in a hdf5 file.

Parameters

- **{object} -- object to dump** (*obj*) –
- **{str} -- name of the hdf5** (*fname*) –

Keyword Arguments **{str} -- root group in the hdf5 file (default** (*root_name*) – {None})

qmctorch.utils.**load_from_hdf5**(*obj*, *fname*, *obj_name*)
Load the content of an hdf5 file in an object.

Parameters

- **{object} -- object where to load the data** (*obj*) –
- **{str} -- name pf the hdf5 file** (*fname*) –
- **{str} -- name of the root group in the hdf5** (*obj_name*) –

qmctorch.utils.**bytes2str**(*bstr*)
Convert a bytes into string.

qmctorch.utils.**register_extra_attributes**(*obj*, *attr_names*)
Register extra attribute to be able to dump them

Parameters

- **{object} -- the object where we want to add attr** (*obj*) –
- **{list} -- a list of attr names** (*attr_names*) –

qmctorch.utils.**fast_power**(*x*, *k*, *mask0=None*, *mask2=None*)
Computes x**k when k have elements 0, 1, 2

Parameters

- **x** (*torch.tensor*) – input
- **k** (*torch.tensor*) – exponents
- **mask0** (*torch.tensor*) – precomputed mask of the elements of that are 0 (Defaults to None and computed here)
- **mask2** (*torch.tensor*) – precomputed mask of the elements of that are 2 (Defaults to None and computed here)

Returns values of x**k

Return type torch.tensor

class qmctorch.utils.**InterpolateMolecularOrbitals**(*wf*)
Bases: object

Interpolation of the AO using a log grid centered on each atom.

get_mo_max_index(*orb*)
Get the index of the highest MO to inlcude in the interpoaltion

Parameters **orb** (*str*) – occupied or all

Raises ValueError – if orb not valid

interpolate_mo_irreg_grid(*pos*, *n*, *orb*)
Interpolate the mo occupied in the configs.

Parameters

- **pos** (*torch.tensor*) – sampling points (Nbatch, 3*Nelec)
- **n** (*int, optional*) – Interpolation order. Defaults to 6.

Returns mo values Nbatch, Nelec, Nmo

Return type torch.tensor

interpolate_mo_reg_grid(*pos*, *res*, *blength*, *orb*)
Interpolate the mo occupied in the configs.

Parameters **pos** (*torch.tensor*) – sampling points (Nbatch, 3*Nelec)

Returns mo values Nbatch, Nelec, Nmo

Return type torch.tensor

class qmctorch.utils.**InterpolateAtomicOrbitals**(*wf*)
Bases: object

Interpolation of the AO using a log grid centered on each atom.

get_interpolator(*n=6*, *length=2*)
evaluate the interpolation function.

Parameters

- **n** (*int, optional*) – number of points on each log axis. Defaults to 6.
- **length** (*int, optional*) – half length of the grid. Defaults to 2.

qmctorch.utils.**btrace**(*M*)
Computes the trace of batched matrices

Parameters **M** (*torch.tensor*) – matrices of size (Na, Nb, ... Nx, N, N)

Returns trace of matrices (Na, Nb, ... Nx)

Return type torch.tensor

Example

```
>>> m = torch.rand(100,5,5)
>>> tr = btrace(m)
```

qmctorch.utils.**bdet2**(*M*)
Computes the determinant of batched 2x2 matrices

Parameters **M** (*torch.tensor*) – input matrices

Returns determinants of the matrices

Return type torch.tensor

qmctorch.utils.**bproj**(*M*, *P*)
Project batched marices using P^T M P

Parameters

- **M** (*torch.tensor*) – batched matrices size (..., N, M)
- **P** (*torch.tensor*) – Porjectors size (..., N, M)

Returns Projected matrices

Return type torch.tensor

qmctorch.utils.**diagonal_hessian**(*out*, *inp*, *return_grads=False*)
return the diagonal hessian of out wrt to inp

Parameters

- **out** (*[type]*) – [description]
- **inp** (*[type]*) – [description]

Returns [description]

Return type [type]

qmctorch.utils.**gradients**(*out*, *inp*)
Return the gradients of out wrt inp

Parameters

- **out** (*[type]*) – [description]
- **inp** (*[type]*) – [description]

14.5 qmctorch.wavefunction package

14.5.1 Subpackages

qmctorch.wavefunction.jastrows package

Subpackages

qmctorch.wavefunction.jastrows.distance package

Submodules

qmctorch.wavefunction.jastrows.distance.electron_electron_distance module

class qmctorch.wavefunction.jastrows.distance.electron_electron_distance.**ElectronElectronDi**

Bases: sphinx.ext.autodoc.importer._MockObject

Computes the electron-electron distances

$$r_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}$$

Parameters

- **nelec** (*int*) – number of electrons
- **ndim** (*int*) – number of spatial dimensions
- **scale** (*bool, optional*) – return scaled values, Defaults to False
- **scale_factor** (*float, optional*) – value of the scale factor, Defaults to 0.6

Examples::

```
>>> edist = ElectronDistance(2,3)
>>> pos = torch.tensor(500,6)
>>> r = edist(pos)
>>> dr = edist(pos,derivative=1)
```

forward(*input*, *derivative=0*)

Compute the pairwise distance between the electrons or its derivative.

When required, the derivative is computed wrt to the first electron i.e.

$$\frac{dr_{ij}}{dx_i}$$

which is different from :

$$\frac{dr_{ij}}{dx_j} = -\frac{dr_{ij}}{dx_i}$$

Parameters

- **input** (*torch.tesnor*) – position of the electron

  size : Nbatch x [Nelec x Ndim]
- **derivative** (*int, optional*) – degre of the derivative.

  Defaults to 0.

Returns

distance (or derivative) matrix

Nbatch x Nelec x Nelec if derivative = 0

Nbatch x Ndim x Nelec x Nelec if derivative = 1,2

Return type torch.tensor

safe_sqrt(*dist*)

Compute the square root of the electron electron distance matrix.

Parameters **dist** (*torch.tensor*) – ee distances squared Nbatch x Nelec x Nelec

Returns sqrt of dist Nbatch x Nelec x Nelec

Return type torch.tensor

get_der_distance(*pos*, *dist*)

Get the derivative of the electron electron distance matrix.

$$\frac{dr_{ij}}{dx_i}$$

Parameters

- **pos** (*torch.tensor*) – positions of the electrons Nbatch x Nelec x Ndim
- **dist** (*torch.tensor*) – distance matrix between the elecs Nbatch x Nelec x Nelec

Returns [description]

Return type [type]

get_second_der_distance(*pos*, *dist*)

Get the second derivative of the electron electron distance matrix.

$$\frac{d^2r_{ij}}{dx_i^2}$$

Parameters

- **pos** (*torch.tensor*) – positions of the electrons Nbatch x Nelec x Ndim
- **dist** (*torch.tensor*) – distance matrix between the elecs Nbatch x Nelec x Nelec

Returns [description]

Return type [type]

static get_distance_quadratic(*pos*)

Compute the distance following a quadratic expansion

Parameters **{torch.tensor} -- electron position [nbatch x nelec x ndim]** (*pos*) –

Returns torch.tensor – distance matrices nbatch x nelec x ndim]

static get_difference(*pos*)

Compute the difference ri - rj

Parameters **{torch.tensor} -- electron position [nbatch x nelec x ndim]** (*pos*) –

Returns torch.tensor – distance matrices nbatch x nelec x nelec x ndim]

qmctorch.wavefunction.jastrows.distance.electron_nuclei_distance module

class qmctorch.wavefunction.jastrows.distance.electron_nuclei_distance.**ElectronNucleiDistar**

Bases: sphinx.ext.autodoc.importer._MockObject

Computes the electron-nuclei distances

$$r_{iA} = \sqrt{(x_i - x_A)^2 + (y_i - y_A)^2 + (z_i - z_A)^2}$$

Parameters

- **nelec** (*int*) – number of electrons
- **atomic_pos** (*tensor*) – positions of the atoms
- **ndim** (*int*) – number of spatial dimensions
- **scale** (*bool, optional*) – return scaled values, Defaults to False
- **scale_factor** (*float, optional*) – value of the scale factor, Defaults to 0.6

Examples::

```
>>> endist = ElectronNucleiDistance(2,2)
>>> epos = torch.tensor(500,6)
>>> r = edist(pos)
>>> dr = edist(pos,derivative=1)
```

forward(*input*, *derivative=0*)

Compute the pairwise distances between electrons and atoms or their derivative.

Parameters

- **input** (*torch.tesnor*) – position of the electron

  size : Nbatch x [Nelec x Ndim]
- **derivative** (*int, optional*) – degre of the derivative.

  Defaults to 0.

Returns

distance (or derivative) matrix

Nbatch x Nelec x Natom if derivative = 0

Nbatch x Ndim x Nelec x Natom if derivative = 1,2

Return type torch.tensor

get_der_distance(*pos*, *dist*)

Get the derivative of the electron-nuclei distance matrix

$$\frac{dr_{iA}}{dx_i}$$

Parameters

- **pos** (*torch.tensor*) – positions of the electrons Nbatch x Nelec x Ndim

- **dist** (*torch.tensor*) – distance matrix between the elecs Nbatch x Nelec x Nelec

Returns [description]

Return type [type]

get_second_der_distance(*pos*, *dist*)
Get the derivative of the electron-nuclei distance matrix

$$\frac{d^2 r_{iA}}{dx_i^2}$$

Parameters

- **pos** (*torch.tensor*) – positions of the electrons Nbatch x Nelec x Ndim
- **dist** (*torch.tensor*) – distance matrix between the elecs Nbatch x Nelec x Nelec

Returns [description]

Return type [type]

qmctorch.wavefunction.jastrows.distance.scaling module

qmctorch.wavefunction.jastrows.distance.scaling.**get_scaled_distance**(*kappa*, *r*)
compute the scaled distance

$$u_{ij} = \frac{1 - e^{-\kappa r_{ij}}}{\kappa}$$

Parameters

- **kappa** (*float*) – scaling factor
- **r** (*torch.tensor*) – matrix of the e-e distances Nbatch x Nelec x Nelec

Returns

values of the scaled distance Nbatch, Nelec, Nelec

Return type torch.tensor

qmctorch.wavefunction.jastrows.distance.scaling.**get_der_scaled_distance**(*kappa*, *r*, *dr*)
Returns the derivative of the scaled distances

$$\frac{du}{dx_i} = \frac{dr_{ij}}{dx_i} e^{-\kappa r_{ij}}$$

Parameters

- **kappa** (*float*) – scaling factor
- **r** (*torch.tensor*) – matrix of the e-e distances Nbatch x Nelec x Nelec
- **dr** (*torch.tensor*) – matrix of the derivative of the e-e distances Nbatch x Ndim x Nelec x Nelec

Returns

deriative of the scaled distance Nbatch x Ndim x Nelec x Nelec

Return type torch.tensor

qmctorch.wavefunction.jastrows.distance.scaling.**get_second_der_scaled_distance**(*kappa*, *r*, *dr*, *d2r*)

computes the second derivative of the scaled distances

$$\frac{d^2u_{ij}}{dx_i^2} = \frac{d^2r_{ij}}{dx_i^2} - \kappa \left(\frac{dr_{ij}}{dx_i}\right)^2 e^{-\kappa r_{ij}}$$

Parameters

- **kappa** (*float*) – scaling factor
- **r** (*torch.tensor*) – unsqueezed matrix of the e-e distances Nbatch x Nelec x Nelec
- **dr** (*torch.tensor*) – matrix of the derivative of the e-e distances Nbatch x Ndim x Nelec x Nelec
- **d2r** (*torch.tensor*) – matrix of the 2nd derivative of the e-e distances Nbatch x Ndim x Nelec x Nelec

Returns

second deriative of the scaled distance Nbatch x Ndim x Nelec x Nelec

Return type torch.tensor

Module contents

qmctorch.wavefunction.jastrows.elec_elec package

Subpackages

qmctorch.wavefunction.jastrows.elec_elec.kernels package

Submodules

qmctorch.wavefunction.jastrows.elec_elec.kernels.fully_connected_jastrow_kernel module

class qmctorch.wavefunction.jastrows.elec_elec.kernels.fully_connected_jastrow_kernel.**Fully**

Bases: *qmctorch.wavefunction.jastrows.elec_elec.kernels.jastrow_kernel_electron_electron_base.JastrowKernelElectronElectronBase*

Defines a fully connected jastrow factors.

get_var_weight()
define the variational weight.

get_static_weight()
Get the matrix of static weights

Returns static weight (0.5 (0.25) for parallel(anti) spins

Return type torch.tensor

forward(*x*)
Compute the kernel values

Parameters **x** (*torch.tensor*) – e-e distance Nbatch, Nele_pairs

Returns values of the f_ij

Return type torch.tensor

qmctorch.wavefunction.jastrows.elec_elec.kernels.jastrow_kernel_electron_electron_base module

class qmctorch.wavefunction.jastrows.elec_elec.kernels.jastrow_kernel_electron_electron_bas

Bases: sphinx.ext.autodoc.importer._MockObject

Base class for the elec-elec jastrow kernels

Parameters

- **nup** (*[type]*) – [description]
- **down** (*[type]*) – [description]
- **cuda** (*bool, optional*) – [description]. Defaults to False.

forward(*r*)
Get the elements of the jastrow matrix :

Parameters **r** (*torch.tensor*) – matrix of the e-e distances Nbatch x Nelec_pair

Returns

matrix fof the jastrow elements Nmo x Nbatch x Nelec_pair

Return type torch.tensor

Note: The kernel receives a [Nbatch x Npair] tensor. The kernel must first reshape that tensor to a [Nbatch*Npair, 1]. The kernel must process this tensor to another [Nbatch*Npair, 1] tensor. The kenrel must reshape the output to a [Nbatch x Npair] tensor.

Example

```
>>> def forward(self, x):
>>>     nbatch, npairs = x.shape
>>>     x = x.reshape(-1, 1)
>>>     x = self.fc1(x)
```

(continues on next page)

(continued from previous page)

```
>>>     ...
>>>     return(x.reshape(nbatch, npairs))
```

compute_derivative(*r*, *dr*)

Get the elements of the derivative of the jastrow kernels wrt to the first electrons using automatic differentiation

Parameters

- **r** (*torch.tensor*) – matrix of the e-e distances Nbatch x Nelec_pair
- **dr** (*torch.tensor*) – matrix of the derivative of the e-e distances Nbatch x Ndim x Nelec_pair

Returns

matrix fof the derivative of the jastrow elements Nmo x Nbatch x Ndim x Nelec_pair

Return type torch.tensor

compute_second_derivative(*r*, *dr*, *d2r*)

Get the elements of the pure 2nd derivative of the jastrow kernels wrt to the first electron using automatic differentiation

Parameters

- **r** (*torch.tensor*) – matrix of the e-e distances Nbatch x Nelec_pair
- **dr** (*torch.tensor*) – matrix of the derivative of the e-e distances Nbatch x Ndim x Nelec_pair
- **d2r** (*torch.tensor*) –

  matrix of the 2nd derivative of the e-e distances

  Nbatch x Ndim x Nelec_pair

Returns

matrix fof the pure 2nd derivative of the jastrow elements Nmo x Nbatch x Ndim x Nelec_pair

Return type torch.tensor

qmctorch.wavefunction.jastrows.elec_elec.kernels.pade_jastrow_kernel module

class qmctorch.wavefunction.jastrows.elec_elec.kernels.pade_jastrow_kernel.**PadeJastrowKerne**

Bases: *qmctorch.wavefunction.jastrows.elec_elec.kernels.jastrow_kernel_electron_electron_base.JastrowKernelElectronElectronBase*

Computes the Simple Pade-Jastrow factor

$$B_{ij} = \frac{w_0 r_{ij}}{1 + w r_{ij}}$$

where w_0 equals 0.5 for parallel spin and 0.25 for antiparallel spin

Parameters

- **nup** (*int*) – number of spin up electons

- **ndow** (*int*) – number of spin down electons
- **cuda** (*bool*) – Turns GPU ON/OFF.
- **w** (*float, optional*) – Value of the variational parameter. Defaults to 1.

get_static_weight()
Get the matrix of static weights

Returns matrix of the static weights

Return type torch.tensor

forward(*r*)
Get the jastrow kernel.

$$B_{ij} = \frac{w_0 r_{i,j}}{1 + w r_{i,j}}$$

Parameters **r** (*torch.tensor*) – matrix of the e-e distances Nbatch x Nelec x Nelec

Returns

matrix of the jastrow kernels Nbatch x Nelec x Nelec

Return type torch.tensor

compute_derivative(*r*, *dr*)
Get the elements of the derivative of the jastrow kernels wrt to the first electrons

$$\frac{dB_{ij}}{dk_i} = \frac{dB_{ij}}{dk_j} = -\frac{dB_{ji}}{dk_i}$$

$$\text{out}_{k,i,j} = A1 + A2$$

$$A1_{kij} = w0 \frac{dr_{ij}}{dk_i} \frac{1}{1 + w r_{ij}}$$

$$A2_{kij} = -w0 w' r_{ij} \frac{dr_{ij}}{dk_i} \frac{1}{1 + w r_{ij}}^2$$

Parameters

- **r** (*torch.tensor*) – matrix of the e-e distances Nbatch x Nelec x Nelec
- **dr** (*torch.tensor*) – matrix of the derivative of the e-e distances Nbatch x Ndim x Nelec x Nelec

Returns

matrix fof the derivative of the jastrow elements Nbatch x Ndim x Nelec x Nelec

Return type torch.tensor

compute_second_derivative(*r*, *dr*, *d2r*)
Get the elements of the pure 2nd derivative of the jastrow kernels wrt to the first electron

$$\frac{d^2 B_{ij}}{dk_i^2} = \frac{d^2 B_{ij}}{dk_j^2} = \frac{d^2 B_{ji}}{dk_i^2}$$

Parameters

- **r** (*torch.tensor*) – matrix of the e-e distances Nbatch x Nelec x Nelec
- **dr** (*torch.tensor*) – matrix of the derivative of the e-e distances Nbatch x Ndim x Nelec x Nelec

- **d2r** (*torch.tensor*) – matrix of the 2nd derivative of the e-e distances Nbatch x Ndim x Nelec x Nelec

Returns

matrix fof the pure 2nd derivative of the jastrow elements Nbatch x Ndim x Nelec x Nelec

Return type torch.tensor

qmctorch.wavefunction.jastrows.elec_elec.kernels.pade_jastrow_polynomial_kernel module

class qmctorch.wavefunction.jastrows.elec_elec.kernels.pade_jastrow_polynomial_kernel.**Pade**

Bases: *qmctorch.wavefunction.jastrows.elec_elec.kernels.jastrow_kernel_electron_electron_base.JastrowKernelElectronElectronBase*

Computes a polynomial Pade-Jastrow factor

$$B_{ij} = \frac{P_{ij}}{Q_{ij}}$$

with : .. math:

```
P_{ij} = a_1 r_{i,j} + a_2 r_{ij}^2 + ....
```

and :

Parameters

- **nup** (*int*) – number of spin up electons
- **ndow** (*int*) – number of spin down electons
- **order** (*int*) – degree of the polynomial
- **weight_a** (*torch.tensor, optional*) – Value of the weight
- **weight_b** (*torch.tensor, optional*) – Value of the weight
- **cuda** (*bool, optional*) – Turns GPU ON/OFF. Defaults to False.

get_static_weight()
Get the matrix of static weights

Returns static weight (0.5 (0.25) for parallel(anti) spins

Return type torch.tensor

set_variational_weights(*weight_a*, *weight_b*)
Define the initial values of the variational weights.

Parameters

- **weight_a** (*torch.tensor or None*) – Value of the weight
- **weight_b** (*torch.tensor or None*) – Value of the weight

forward(*r*)
Get the jastrow kernel.

$$B_{ij} = \frac{P_{ij}}{Q_{ij}}$$

Parameters **r** (*torch.tensor*) – matrix of the e-e distances Nbatch x Nelec x Nelec

Returns

matrix of the jastrow kernels Nbatch x Nelec x Nelec

Return type torch.tensor

compute_derivative(*r*, *dr*)
Get the elements of the derivative of the jastrow kernels wrt to the first electrons

The derivative is given by:

$$\text{out}_{k,i,j} = \frac{P'Q - PQ'}{Q^2}$$

with:

$$P_{ij} = a_1 r_{i,j} + a_2 r_{ij}^2 + ...Q_{ij} = 1 + b_1 r_{i,j} + b_2 r_{ij}^2 +$$

and :

$$P'_{ij} = a_1 dr + a_2 2rdr + a_r 3drr^2 + ...Q'_{ij} = b_1 dr + b_2 2rdr + b_r 3drr^2 + ...$$

Due to the properties of the derivative we have .. math:

```
\frac{d B_{ij}}{d k_i} =  \frac{d B_{ij}}{d k_j}  = -\frac{d B_{ji}{d k_i}
```

Parameters

- **r** (*torch.tensor*) – matrix of the e-e distances Nbatch x Nelec x Nelec
- **dr** (*torch.tensor*) – matrix of the derivative of the e-e distances Nbatch x Ndim x Nelec x Nelec

Returns

matrix fof the derivative of the jastrow elements Nbatch x Ndim x Nelec x Nelec

Return type torch.tensor

compute_second_derivative(*r*, *dr*, *d2r*)
Get the elements of the pure 2nd derivative of the jastrow kernels wrt to the first electron

Due to the properties of the derivative we have .. math:

```
\frac{d B_{ij}}{d k_i} =  \frac{d B_{ij}}{d k_j}  = \frac{d B_{ji}{d k_i}
```

Parameters

- **r** (*torch.tensor*) – matrix of the e-e distances Nbatch x Nelec x Nelec
- **dr** (*torch.tensor*) – matrix of the derivative of the e-e distances Nbatch x Ndim x Nelec x Nelec
- **d2r** (*torch.tensor*) –

matrix of the 2nd derivative of the e-e distances

Nbatch x Ndim x Nelec x Nelec

Returns

matrix fof the pure 2nd derivative of the jastrow elements Nbatch x Ndim x Nelec x Nelec

Return type torch.tensor

Module contents

Submodules

qmctorch.wavefunction.jastrows.elec_elec.jastrow_factor_electron_electron module

class qmctorch.wavefunction.jastrows.elec_elec.jastrow_factor_electron_electron.**JastrowFact**

Bases: sphinx.ext.autodoc.importer._MockObject

Electron-Electron Jastrow factor.

$$J = \prod_{i<j} \exp(\text{Kernel}(r_{ij}))$$

Parameters

- **nup** (*int*) – number of spin up electons
- **ndow** (*int*) – number of spin down electons
- **jastrow_kernel** (*kernel*) – class of a electron-electron Jastrow kernel
- **kernel_kwargs** (*dict, optional*) – keyword argument of the kernel. Defaults to {}.
- **orbital_dependent_kernel** (*bool, optional*) – Make the kernel orbital dependent. Defaults to False.
- **number_of_orbitals** (*int, optional*) – number of orbitals for orbital dependent kernels. Defaults to None.
- **scale** (*bool, optional*) – use scaled electron-electron distance. Defaults to False.
- **scale_factor** (*float, optional*) – scaling factor. Defaults to 0.6.
- **cuda** (*bool, optional*) – use cuda. Defaults to False.

get_mask_tri_up()

Get the mask to select the triangular up matrix

Returns mask of the tri up matrix

Return type torch.tensor

extract_tri_up(*inp*)
extract the upper triangular elements

Parameters **inp** (*torch.tensor*) – input matrices (nbatch, n, n)

Returns triangular up element (nbatch, -1)

Return type torch.tensor

get_edist_unique(*pos*, *derivative=0*)
Get the unique elements of the electron-electron distance matrix.

Parameters

- **pos** (*torch.tensor*) – positions of the electrons (Nbatch, Nelec*3)
- **derivative** (*int, optional*) – order of the derivative

Returns unique values of the electron-electron distance matrix

Return type torch.tensor

forward(*pos*, *derivative=0*, *sum_grad=True*)
Compute the Jastrow factors.

Parameters

- **pos** (*torch.tensor*) – Positions of the electrons Size : Nbatch, Nelec x Ndim
- **derivative** (*int, optional*) – order of the derivative (0,1,2,). Defaults to 0.
- **sum_grad** (*bool, optional*) – Return the sum_grad (i.e. the sum of the derivatives) or the individual terms. Defaults to True. False only for derivative=1

Returns

value of the jastrow parameter for all confs derivative = 0 (Nmo) x Nbatch x 1 derivative = 1 (Nmo) x Nbatch x Nelec (for sum_grad = True) derivative = 1 (Nmo) x Nbatch x Ndim x Nelec (for sum_grad = False) derivative = 2 (Nmo) x Nbatch x Nelec

Return type torch.tensor

jastrow_factor_derivative(*r*, *dr*, *jast*, *sum_grad*)
Compute the value of the derivative of the Jastrow factor

Parameters

- **r** (*torch.tensor*) – distance matrix Nbatch x Nelec x Nelec
- **dr** (*torch.tensor*) – derivative of the distance matrix Nbatch x Nelec x Nelec x 3
- **jast** (*torch.tensor*) – values of the jastrow elements Nbatch x Nelec x Nelec
- **sum_grad** (*bool*) – return the sum of the gradient or the individual components

Returns

gradient of the jastrow factors Nbatch x Nelec x Ndim

Return type torch.tensor

jastrow_factor_second_derivative(*r*, *dr*, *d2r*, *jast*)
Compute the value of the pure 2nd derivative of the Jastrow factor

Parameters

- **r** (*torch.tensor*) – distance matrix Nbatch x Nelec x Nelec
- **dr** (*torch.tensor*) – derivative of the distance matrix Nbatch x Nelec x Nelec x 3
- **d2r** (*torch.tensor*) – 2nd derivative of the distance matrix Nbatch x Nelec x Nelec x 3
- **jast** (*torch.tensor*) – values of the ajstrow elements Nbatch x Nelec x Nelec

Returns

diagonal hessian of the jastrow factors Nbatch x Nelec x Ndim

Return type torch.tensor

partial_derivative(*djast*)

Computes the partial derivative

Parameters **djast** (*torch.tensor*) – values of the derivative of the jastrow kernels

qmctorch.wavefunction.jastrows.elec_elec.orbital_dependent_jastrow_kernel module

class qmctorch.wavefunction.jastrows.elec_elec.orbital_dependent_jastrow_kernel.**OrbitalDepe**

Bases: *qmctorch.wavefunction.jastrows.elec_elec.kernels.jastrow_kernel_electron_electron_base.JastrowKernelElectronElectronBase*

Transform a kernel into a orbital dependent kernel

Parameters

- **nup** (*int*) – number of spin up electrons
- **ndown** (*int*) – number of spin down electron
- **nmo** (*int*) – number of orbital
- **cuda** (*bool*) – use GPUs
- **jastrow_kernel** (*kernel function*) – kernel to be used
- **kernel_kwargs** (*dict*) – keyword arguments of the kernel

forward(*r*)

Get the jastrow kernel.

Parameters **r** (*torch.tensor*) – matrix of the e-e distances Nbatch x Nelec_pair

Returns

matrix of the jastrow kernels Nmo x Nbatch x Nelec_pair

Return type torch.tensor

compute_derivative(*r*, *dr*)

Get the elements of the derivative of the jastrow kernels wrt to the first electrons

Parameters

- **r** (*torch.tensor*) – matrix of the e-e distances Nbatch x Nelec_pair
- **dr** (*torch.tensor*) – matrix of the derivative of the e-e distances Nbatch x Ndim x Nelec_pair

Returns

matrix fof the derivative of the jastrow elements Nmo x Nbatch x Ndim x Nelec_pair

Return type torch.tensor

compute_second_derivative(*r*, *dr*, *d2r*)

Get the elements of the pure 2nd derivative of the jastrow kernels wrt to the first electron

Parameters

- **r** (*torch.tensor*) – matrix of the e-e distances Nbatch x Nelec_pair
- **dr** (*torch.tensor*) – matrix of the derivative of the e-e distances Nbatch x Ndim x Nelec_pair
- **d2r** (*torch.tensor*) –

  matrix of the 2nd derivative of the e-e distances

  Nbatch x Ndim x Nelec_pair

Returns

matrix fof the pure 2nd derivative of the jastrow elements Nmo x Nbatch x Ndim x Nelec_pair

Return type torch.tensor

Module contents

qmctorch.wavefunction.jastrows.elec_elec_nuclei package

Subpackages

qmctorch.wavefunction.jastrows.elec_elec_nuclei.kernels package

Submodules

qmctorch.wavefunction.jastrows.elec_elec_nuclei.kernels.fully_connected_jastrow_kernel module

class qmctorch.wavefunction.jastrows.elec_elec_nuclei.kernels.fully_connected_jastrow_kerne

Bases: *qmctorch.wavefunction.jastrows.elec_elec_nuclei.kernels.jastrow_kernel_electron_electron_nuclei_base.JastrowKernelElectronElectronNucleiBase*

Defines a fully connected jastrow factors.

forward(*x*)

Compute the values of the individual f_ij=f(r_ij)

Parameters **x** (*torch.tensor*) – e-e distance Nbatch, Nele_pairs

Returns values of the f_ij

Return type torch.tensor

qmctorch.wavefunction.jastrows.elec_elec_nuclei.kernels.jastrow_kernel_electron_electron_nuclei_base module

class qmctorch.wavefunction.jastrows.elec_elec_nuclei.kernels.jastrow_kernel_electron_elect

Bases: sphinx.ext.autodoc.importer._MockObject

Base Class for the elec-elec-nuc jastrow kernel

Parameters

- **nup** (*int*) – number of spin up electons
- **ndow** (*int*) – number of spin down electons
- **atoms** (*torch.tensor*) – atomic positions of the atoms
- **cuda** (*bool, optional*) – Turns GPU ON/OFF. Defaults to False.

forward(*x*)

Compute the values of the kernel

Parameters **x** (*torch.tensor*) – e-e and e-n distances distance (Nbatch, Natom, Nelec_pairs, 3) the last dimension holds the values [R_{iA}, R_{jA}, r_{ij}] in that order.

Returns values of the kernel (Nbatch, Natom, Nelec_pairs, 1)

Return type torch.tensor

compute_derivative(*r*, *dr*)

Get the elements of the derivative of the jastrow kernels.

compute_second_derivative(*r*, *dr*, *d2r*)

Get the elements of the pure 2nd derivative of the jastrow kernels.

Module contents

Submodules

qmctorch.wavefunction.jastrows.elec_elec_nuclei.jastrow_factor_electron_electron_nuclei module

class qmctorch.wavefunction.jastrows.elec_elec_nuclei.jastrow_factor_electron_electron_nucl

Bases: sphinx.ext.autodoc.importer._MockObject

Jastrow Factor of the elec-elec-nuc term:

$$J = \exp\left(\sum_A \sum_{i<j} K(R_{iA}, r_{jA}, r_{rij})\right)$$

Parameters

- **nup** (*int*) – number of spin up electons
- **ndow** (*int*) – number of spin down electons
- **cuda** (*bool, optional*) – Turns GPU ON/OFF. Defaults to False.

get_mask_tri_up()
Get the mask to select the triangular up matrix

Returns mask of the tri up matrix

Return type torch.tensor

extract_tri_up(*inp*)
extract the upper triangular elements

Parameters **input** (*torch.tensor*) – input matrices (..., nelec, nelec)

Returns triangular up element (..., nelec_pair)

Return type torch.tensor

extract_elec_nuc_dist(*en_dist*)
Organize the elec nuc distances

Parameters **en_dist** (*torch.tensor*) – electron-nuclei distances nbatch x nelec x natom or nbatch x 3 x nelec x natom (dr)

Returns nbatch x natom x nelec_pair x 2 or torch.tensor: nbatch x 3 x natom x nelec_pair x 2 (dr)

Return type torch.tensor

assemble_dist(*pos*)
Assemle the different distances for easy calculations

Parameters **pos** (*torch.tensor*) – Positions of the electrons Size : Nbatch, Nelec x Ndim

Returns nbatch, natom, nelec_pair, 3

Return type torch.tensor

assemble_dist_deriv(*pos*, *derivative=1*)

Assemle the different distances for easy calculations the output has dimension nbatch, 3 x natom, nelec_pair, 3 the last dimension is composed of [r_{e_1n}, r_{e_2n}, r_{ee}]

Parameters **pos** (*torch.tensor*) – Positions of the electrons Size : Nbatch, Nelec x Ndim

Returns nbatch, 3 x natom, nelec_pair, 3

Return type torch.tensor

forward(*pos*, *derivative=0*, *sum_grad=True*)
Compute the Jastrow factors.

Parameters

- **pos** (*torch.tensor*) – Positions of the electrons Size : Nbatch, Nelec x Ndim

- **derivative** (*int, optional*) – order of the derivative (0,1,2,). Defaults to 0.
- **sum_grad** (*bool, optional*) – Return the sum_grad (i.e. the sum of the derivatives) or the individual terms. Defaults to True. False only for derivative=1

Returns

value of the jastrow parameter for all confs derivative = 0 (Nmo) x Nbatch x 1 derivative = 1 (Nmo) x Nbatch x Nelec (for sum_grad = True) derivative = 1 (Nmo) x Nbatch x Ndim x Nelec (for sum_grad = False) derivative = 2 (Nmo) x Nbatch x Nelec

Return type torch.tensor

jastrow_factor_derivative(*r*, *dr*, *jast*, *sum_grad*)
Compute the value of the derivative of the Jastrow factor

Parameters

- **r** (*torch.tensor*) – ee distance matrix Nbatch x Nelec x Nelec
- **jast** (*torch.tensor*) – values of the jastrow elements Nbatch x Nelec x Nelec

Returns

gradient of the jastrow factors Nbatch x Nelec x Ndim

Return type torch.tensor

jastrow_factor_second_derivative(*r*, *dr*, *d2r*, *jast*)
Compute the value of the pure 2nd derivative of the Jastrow factor

Parameters

- **r** (*torch.tensor*) – ee distance matrix Nbatch x Nelec x Nelec
- **jast** (*torch.tensor*) – values of the ajstrow elements Nbatch x Nelec x Nelec

Returns

diagonal hessian of the jastrow factors Nbatch x Nelec x Ndim

Return type torch.tensor

partial_derivative(*djast*)
[summary]

Parameters **djast** (*[type]*) – [description]

jastrow_factor_second_derivative_auto(*pos*, *jast=None*)
Compute the second derivative of the jastrow factor automatically. This is needed for complicate kernels where the partial derivatives of the kernels are difficult to organize in a total derivaitve e.e Boys-Handy

Parameters **pos** (*[type]*) – [description]

Module contents

qmctorch.wavefunction.jastrows.elec_nuclei package

Subpackages

qmctorch.wavefunction.jastrows.elec_nuclei.kernels package

Submodules

qmctorch.wavefunction.jastrows.elec_nuclei.kernels.fully_connected_jastrow_kernel module

class qmctorch.wavefunction.jastrows.elec_nuclei.kernels.fully_connected_jastrow_kernel.**Ful**

Bases: *qmctorch.wavefunction.jastrows.elec_nuclei.kernels.jastrow_kernel_electron_nuclei_base.JastrowKernelElectronNucleiBase*

Computes the Simple Pade-Jastrow factor

$$J = \prod_{i<j} \exp(B_{ij})$$
$$textwith \qquad B_{ij} =$$
$$fracw_0 r_{i,j} 1 + w r_{i,j}$$

Parameters

- **nup** (*int*) – number of spin up electons
- **ndow** (*int*) – number of spin down electons
- **atoms** (*torch.tensor*) – atomic positions of the atoms
- **w** (*float, optional*) – Value of the variational parameter. Defaults to 1..
- **cuda** (*bool, optional*) – Turns GPU ON/OFF. Defaults to False.

forward(*x*)

Get the jastrow kernel.

Parameters **x** (*torch.tensor*) – matrix of the e-e distances Nbatch x Nelec x Nnuc

Returns

matrix of the jastrow kernels Nbatch x Nelec x Nnuc

Return type torch.tensor

qmctorch.wavefunction.jastrows.elec_nuclei.kernels.jastrow_kernel_electron_nuclei_base module

class qmctorch.wavefunction.jastrows.elec_nuclei.kernels.jastrow_kernel_electron_nuclei_bas

Bases: sphinx.ext.autodoc.importer._MockObject

Base class for the elec-nuc jastrow factor

Parameters

- **nup** (*int*) – number of spin up electons
- **ndow** (*int*) – number of spin down electons
- **atoms** (*torch.tensor*) – atomic positions of the atoms

- **w** (*float, optional*) – Value of the variational parameter. Defaults to 1..
- **cuda** (*bool, optional*) – Turns GPU ON/OFF. Defaults to False.

forward(*r*)

Get the elements of the jastrow matrix : .. math:

```
out_{i,j} = \exp{ \frac{b r_{i,j}}{1+b'r_{i,j}} }
```

Parameters **r** (*torch.tensor*) – matrix of the e-e distances Nbatch x Nelec x Nelec

Returns

matrix fof the jastrow elements Nbatch x Nelec x Nelec

Return type torch.tensor

compute_derivative(*r*, *dr*)

Get the elements of the derivative of the jastrow kernels wrt to the first electrons

$$dB_{ij}/dk_i = dB_{ij}/dk_j = -dB_{ji}/dk_i$$
$$out_{k,i,j} = A1 + A2A1_{kij} = w0$$

rac{dr_{ij}}{dk_i} / (1 + w r_{ij}) A2_{kij} = - w0 w' r_{ij}

rac{dr_{ij}}{dk_i} / (1 + w r_{ij})^2

Args:

r (torch.tensor): matrix of the e-e distances Nbatch x Nelec x Nelec

dr (torch.tensor): matrix of the derivative of the e-e distances Nbatch x Ndim x Nelec x Nelec

Returns:

torch.tensor: matrix fof the derivative of the jastrow elements Nbatch x Ndim x Nelec x Nelec

compute_second_derivative(*r*, *dr*, *d2r*)

Get the elements of the pure 2nd derivative of the jastrow kernels wrt to the first electron

$$d^2B_{ij}/dk_i^2 = d^2B_{ij}/dk_j^2 = d^2B_{ji}/dk_i^2$$

Parameters

- **r** (*torch.tensor*) – matrix of the e-e distances Nbatch x Nelec x Nelec
- **dr** (*torch.tensor*) – matrix of the derivative of the e-e distances Nbatch x Ndim x Nelec x Nelec
- **d2r** (*torch.tensor*) –

  matrix of the 2nd derivative of the e-e distances

  Nbatch x Ndim x Nelec x Nelec

Returns

matrix fof the pure 2nd derivative of the jastrow elements Nbatch x Ndim x Nelec x Nelec

Return type torch.tensor

qmctorch.wavefunction.jastrows.elec_nuclei.kernels.pade_jastrow_kernel module

class qmctorch.wavefunction.jastrows.elec_nuclei.kernels.pade_jastrow_kernel.**PadeJastrowKe**

Bases: *qmctorch.wavefunction.jastrows.elec_nuclei.kernels.jastrow_kernel_electron_nuclei_base.JastrowKernelElectronNucleiBase*

Computes the Simple Pade-Jastrow factor

$$J = \prod_{i<j} \exp(B_{ij})$$
$$textwith \qquad B_{ij} =$$
$$fracw_0 r_{i,j} 1 + w r_{i,j}$$

Parameters

- **nup** (*int*) – number of spin up electons
- **ndow** (*int*) – number of spin down electons
- **atoms** (*torch.tensor*) – atomic positions of the atoms
- **w** (*float, optional*) – Value of the variational parameter. Defaults to 1..
- **cuda** (*bool, optional*) – Turns GPU ON/OFF. Defaults to False.

forward(*r*)

Get the jastrow kernel.

$$B_{ij} =$$

rac{b r_{i,j}}{1+b'r_{i,j}}

Args:

r (torch.tensor): matrix of the e-e distances Nbatch x Nelec x Nelec

Returns:

torch.tensor: matrix of the jastrow kernels Nbatch x Nelec x Nelec

compute_derivative(*r*, *dr*)

Get the elements of the derivative of the jastrow kernels wrt to the first electrons

$$dB_{ij}/dk_i = dB_{ij}/dk_j = -dB_{ji}/dk_i$$
$$out_{k,i,j} = A1 + A2A1_{kij} = w0$$

rac{dr_{ij}}{dk_i} / (1 + w r_{ij}) A2_{kij} = - w0 w' r_{ij}

rac{dr_{ij}}{dk_i} / (1 + w r_{ij})^2

Args:

r (torch.tensor): matrix of the e-e distances Nbatch x Nelec x Nelec

dr (torch.tensor): matrix of the derivative of the e-e distances Nbatch x Ndim x Nelec x Nelec

Returns:

torch.tensor: matrix fof the derivative of the jastrow elements Nbatch x Ndim x Nelec x Nelec

compute_second_derivative(*r*, *dr*, *d2r*)

Get the elements of the pure 2nd derivative of the jastrow kernels wrt to the first electron

$$d^2B_{ij}/dk_i^2 = d^2B_{ij}/dk_j^2 = d^2B_{ji}/dk_i^2$$

Parameters

- **r** (*torch.tensor*) – matrix of the e-e distances Nbatch x Nelec x Nelec
- **dr** (*torch.tensor*) – matrix of the derivative of the e-e distances Nbatch x Ndim x Nelec x Nelec
- **d2r** (*torch.tensor*) –

  matrix of the 2nd derivative of the e-e distances

  Nbatch x Ndim x Nelec x Nelec

Returns

matrix fof the pure 2nd derivative of the jastrow elements Nbatch x Ndim x Nelec x Nelec

Return type torch.tensor

Module contents

Submodules

qmctorch.wavefunction.jastrows.elec_nuclei.jastrow_factor_electron_nuclei module

class qmctorch.wavefunction.jastrows.elec_nuclei.jastrow_factor_electron_nuclei.**JastrowFact**

Bases: sphinx.ext.autodoc.importer._MockObject

Base class for two el-nuc jastrow of the form:

$$J = \prod_{a,i} \exp(A(r_{ai}))$$

Parameters

- **nup** (*int*) – number of spin up electons
- **ndow** (*int*) – number of spin down electons
- **atomic_pos** (*tensor*) – positions of the atoms
- **cuda** (*bool, optional*) – Turns GPU ON/OFF. Defaults to False.

forward(*pos*, *derivative=0*, *sum_grad=True*)

Compute the Jastrow factors.

Parameters

- **pos** (*torch.tensor*) – Positions of the electrons Size : Nbatch, Nelec x Ndim
- **derivative** (*int, optional*) – order of the derivative (0,1,2,). Defaults to 0.
- **sum_grad** (*bool, optional*) – Return the sum_grad (i.e. the sum of the derivatives) or the individual terms. Defaults to True. False only for derivative=1

Returns

value of the jastrow parameter for all confs derivative = 0 (Nmo) x Nbatch x 1 derivative = 1 (Nmo) x Nbatch x Nelec (for sum_grad = True) derivative = 1 (Nmo) x Nbatch x Ndim x Nelec (for sum_grad = False) derivative = 2 (Nmo) x Nbatch x Nelec

Return type torch.tensor

jastrow_factor_derivative(*r*, *dr*, *jast*, *sum_grad*)

Compute the value of the derivative of the Jastrow factor

Parameters

- **r** (*torch.tensor*) – ee distance matrix Nbatch x Nelec x Nelec
- **jast** (*torch.tensor*) – values of the jastrow elements Nbatch x Nelec x Natom

Returns

gradient of the jastrow factors Nbatch x Ndim x Nelec

Return type torch.tensor

jastrow_factor_second_derivative(*r*, *dr*, *d2r*, *jast*)

Compute the value of the pure 2nd derivative of the Jastrow factor

Parameters

- **r** (*torch.tensor*) – ee distance matrix Nbatch x Nelec x Nelec
- **jast** (*torch.tensor*) – values of the ajstrow elements Nbatch x Nelec x Nelec

Returns

diagonal hessian of the jastrow factors Nbatch x Nelec x Ndim

Return type torch.tensor

Module contents

Submodules

qmctorch.wavefunction.jastrows.jastrow_factor_combined_terms module

class qmctorch.wavefunction.jastrows.jastrow_factor_combined_terms.**JastrowFactorCombinedTe**

Bases: sphinx.ext.autodoc.importer._MockObject

[summary]

Parameters

- **nup** (*int*) – number of spin up electron
- **ndown** (*int*) – number opf spin down electron
- **atomic_pos** (*torch tensor*) – atomic positions
- **jastrow_kernel** (*[dict]*) – kernels of the jastrow factor
- **jastrow_kernel_kwargs** (*dict*) – keyword argument of the kernels
- **cuda** (*bool, optional*) – [description]. Defaults to False.

forward(*pos*, *derivative=0*, *sum_grad=True*)

Compute the Jastrow factors.

Parameters

- **pos** (*torch.tensor*) – Positions of the electrons Size: Nbatch, Nelec x Ndim
- **derivative** (*int, optional*) – order of the derivative (0, 1, 2,). Defaults to 0.
- **sum_grad** (*bool, optional*) –

  Return the sum_grad(i.e. the sum of the derivatives)

  terms. Defaults to True. False only for derivative = 1

Returns

value of the jastrow parameter for all confs

derivative = 0 (Nmo) x Nbatch x 1

derivative = 1 (Nmo) x Nbatch x Nelec

(for sum_grad = True)

derivative = 1 (Nmo) x Nbatch x Ndim x Nelec (for sum_grad = False)

Return type torch.tensor

static get_combined_values(*jast_vals*)
Compute the product of all terms in jast_vals.

static get_derivative_combined_values(*jast_vals*, *djast_vals*)
Compute the derivative of the product. .. math:

J = A * B * C frac{d J}{dx} = frac{d A}{dx} B C + A frac{d B}{dx} C + A B frac{d C}{dx}

static get_second_derivative_combined_values(*jast_vals*, *djast_vals*, *d2jast_vals*)
Compute the derivative of the product. .. math:

J = A * B * C frac{d^2 J}{dx^2} = frac{d^2 A}{dx^2} B C + A frac{d^2 B}{dx^2} C + A B frac{d^2 C}{dx^2}

- 2(frac{d A}{dx} frac{dB}{dx} C + frac{d A}{dx} B frac{dC}{dx} + A frac{d B}{dx} frac{dC}{dx})

Module contents

qmctorch.wavefunction.orbitals package

Subpackages

qmctorch.wavefunction.orbitals.backflow package

Subpackages

qmctorch.wavefunction.orbitals.backflow.kernels package

Submodules

qmctorch.wavefunction.orbitals.backflow.kernels.backflow_kernel_autodiff_inverse module

class qmctorch.wavefunction.orbitals.backflow.kernels.backflow_kernel_autodiff_inverse.**Back**

Bases: *qmctorch.wavefunction.orbitals.backflow.kernels.backflow_kernel_base.BackFlowKernelBase*

Compute the back flow kernel, i.e. the function f(rij) where rij is the distance between electron i and j This kernel is used in the backflow transformation .. math:
q_i = r_i + sum_{jneq i} f(r_{ij}) (r_i-r_j)

qmctorch.wavefunction.orbitals.backflow.kernels.backflow_kernel_base module

class qmctorch.wavefunction.orbitals.backflow.kernels.backflow_kernel_base.**BackFlowKernelBa**

Bases: sphinx.ext.autodoc.importer._MockObject

Compute the back flow kernel, i.e. the function f(rij) where rij is the distance between electron i and j This kernel is used in the backflow transformation .. math:
q_i = r_i + sum_{jneq i} f(r_{ij}) (r_i-r_j)

forward(*ree*, *derivative=0*)

Computes the desired values of the kernel

Args: ree (torch.tensor): e-e distance Nbatch x Nelec x Nelec derivative (int): derivative requried 0, 1, 2

Returns f(r) Nbatch x Nelec x Nelec

Return type torch.tensor

qmctorch.wavefunction.orbitals.backflow.kernels.backflow_kernel_fully_connected module

class qmctorch.wavefunction.orbitals.backflow.kernels.backflow_kernel_fully_connected.**Backl**

Bases: *qmctorch.wavefunction.orbitals.backflow.kernels.backflow_kernel_base.BackFlowKernelBase*

Compute the back flow kernel, i.e. the function f(rij) where rij is the distance between electron i and j This kernel is used in the backflow transformation .. math:
q_i = r_i + sum_{jneq i} f(r_{ij}) (r_i-r_j)

qmctorch.wavefunction.orbitals.backflow.kernels.backflow_kernel_inverse module

class qmctorch.wavefunction.orbitals.backflow.kernels.backflow_kernel_inverse.**BackFlowKerne**

Bases: *qmctorch.wavefunction.orbitals.backflow.kernels.backflow_kernel_base.BackFlowKernelBase*

Compute the back flow kernel, i.e. the function f(rij) where rij is the distance between electron i and j This kernel is used in the backflow transformation .. math:
q_i = r_i + sum_{jneq i} f(r_{ij}) (r_i-r_j)
with here :

qmctorch.wavefunction.orbitals.backflow.kernels.backflow_kernel_power_sum module

class qmctorch.wavefunction.orbitals.backflow.kernels.backflow_kernel_power_sum.**BackFlowKe**

Bases: *qmctorch.wavefunction.orbitals.backflow.kernels.backflow_kernel_base.BackFlowKernelBase*

Compute the back flow kernel, i.e. the function f(rij) where rij is the distance between electron i and j This kernel is used in the backflow transformation .. math:
q_i = r_i + sum_{jneq i} f(r_{ij}) (r_i-r_j)

qmctorch.wavefunction.orbitals.backflow.kernels.backflow_kernel_square module

class qmctorch.wavefunction.orbitals.backflow.kernels.backflow_kernel_square.**BackFlowKernel**

Bases: *qmctorch.wavefunction.orbitals.backflow.kernels.backflow_kernel_base.BackFlowKernelBase*

Define a generic kernel to test the auto diff features.

Module contents

Submodules

qmctorch.wavefunction.orbitals.backflow.backflow_transformation module

class qmctorch.wavefunction.orbitals.backflow.backflow_transformation.**BackFlowTransformati**

Bases: sphinx.ext.autodoc.importer._MockObject

Transform the electorn coordinates into backflow coordinates. see : Orbital-dependent backflow wave functions for real-space quantum Monte Carlo https://arxiv.org/abs/1910.07167
eq i} eta(r_{ij})(bold{r}_i - bold{r}_j)

forward(*pos*, *derivative=0*)

qmctorch.wavefunction.orbitals.backflow.orbital_dependent_backflow_kernel module

class qmctorch.wavefunction.orbitals.backflow.orbital_dependent_backflow_kernel.**OrbitalDepe**

Bases: sphinx.ext.autodoc.importer._MockObject

Compute orbital dependent back flow kernel, i.e. the functions f(rij) where rij is the distance between electron i and j This kernel is used in the backflow transformation .. math:
q^{alpha}_i = r_i + sum_{jneq i} f^{alpha}(r_{ij}) (r_i-r_j)
where :math: *f^{alpha}(r_{ij})* is the kernel for obital :math: *alpha*

forward(*ree*, *derivative=0*)

Computes the desired values of the kernels

Args: ree (torch.tensor): e-e distance Nbatch x Nelec x Nelec derivative (int): derivative requried 0, 1, 2

Returns f(r) Nbatch x Nao x Nelec x Nelec

Return type torch.tensor

qmctorch.wavefunction.orbitals.backflow.orbital_dependent_backflow_transformation module

class qmctorch.wavefunction.orbitals.backflow.orbital_dependent_backflow_transformation.**Orb**

Bases: sphinx.ext.autodoc.importer._MockObject

Transform the electorn coordinates into backflow coordinates. see : Orbital-dependent backflow wave functions for real-space quantum Monte Carlo https://arxiv.org/abs/1910.07167
eq i} eta(r_{ij})(bold{r}_i - bold{r}_j)

forward(*pos*, *derivative=0*)

Module contents

Submodules

qmctorch.wavefunction.orbitals.atomic_orbitals module

class qmctorch.wavefunction.orbitals.atomic_orbitals.**AtomicOrbitals**(*mol*, *cuda=False*)

Bases: sphinx.ext.autodoc.importer._MockObject

Computes the value of atomic orbitals

Parameters

- **mol** (Molecule) – Molecule object
- **cuda** (*bool, optional*) – Turn GPU ON/OFF Defaults to False.

forward(*pos, derivative=[0], sum_grad=True, sum_hess=True, one_elec=False*)

Computes the values of the atomic orbitals.

$$\phi_i(r_j) = \sum_n c_n \mathrm{Rad}^i_n(r_j)\mathrm{Y}^i_n(r_j)$$

where Rad is the radial part and Y the spherical harmonics part. It is also possible to compute the first and second derivatives

$$\nabla\phi_i(r_j) = \frac{d}{dx_j}\phi_i(r_j) + \frac{d}{dy_j}\phi_i(r_j) + \frac{d}{dz_j}\phi_i(r_j)$$
$$\mathrm{grad}\phi_i(r_j) = (\frac{d}{dx_j}\phi_i(r_j), \frac{d}{dy_j}\phi_i(r_j), \frac{d}{dz_j}\phi_i(r_j))$$
$$\Delta\phi_i(r_j) = \frac{d^2}{dx_j^2}\phi_i(r_j) + \frac{d^2}{dy_j^2}\phi_i(r_j) + \frac{d^2}{dz_j^2}\phi_i(r_j)$$

Parameters

- **pos** (*torch.tensor*) – Positions of the electrons Size : Nbatch, Nelec x Ndim
- **derivative** (*int, optional*) – order of the derivative (0,1,2,). Defaults to 0.
- **sum_grad** (*bool, optional*) – Return the sum_grad (i.e. the sum of the derivatives) or the individual terms. Defaults to True. False only for derivative=1

- **sum_hess** (*bool, optional*) – Return the sum_hess (i.e. the sum of 2nd the derivatives) or the individual terms. Defaults to True. False only for derivative=1
- **one_elec** (*bool, optional*) – if only one electron is in input

Returns

Value of the AO (or their derivatives)

size : Nbatch, Nelec, Norb (sum_grad = True)

size : Nbatch, Nelec, Norb, Ndim (sum_grad = False)

Return type torch.tensor

Examples::

```
>>> mol = Molecule('h2.xyz')
>>> ao = AtomicOrbitals(mol)
>>> pos = torch.rand(100,6)
>>> aovals = ao(pos)
>>> daovals = ao(pos,derivative=1)
```

update (*ao*, *pos*, *idelec*)

Update an AO matrix with the new positions of one electron

Parameters

- **ao** (*torch.tensor*) – initial AO matrix
- **pos** (*torch.tensor*) – new positions of some electrons
- **idelec** (*int*) – index of the electron that has moved

Returns new AO matrix

Return type torch.tensor

Examples::

```
>>> mol = Molecule('h2.xyz')
>>> ao = AtomicOrbitals(mol)
>>> pos = torch.rand(100,6)
>>> aovals = ao(pos)
>>> id = 0
>>> pos[:,:3] = torch.rand(100,3)
>>> ao.update(aovals, pos, 0)
```

qmctorch.wavefunction.orbitals.atomic_orbitals_backflow module

class qmctorch.wavefunction.orbitals.atomic_orbitals_backflow.**AtomicOrbitalsBackFlow**(*mol*, *back-flow_ke*
back-flow_ke
cuda=F

Bases: *qmctorch.wavefunction.orbitals.atomic_orbitals.AtomicOrbitals*

Computes the value of atomic orbitals

Parameters

- **mol** (Molecule) – Molecule object
- **cuda** (*bool, optional*) – Turn GPU ON/OFF Defaults to False.

forward (*pos, derivative=[0], sum_grad=True, sum_hess=True, one_elec=False*)

Computes the values of the atomic orbitals.

$$\phi_i(q_j) = \sum_n c_n \mathrm{Rad}^i_n(q_j) \mathrm{Y}^i_n(q_j)$$

where :math: *text{Rad}^{i}_n(r_j)* is the radial part and :math: *text{Y}^{i}_n(r_j)* the spherical harmonics part.

The electronic positions are calculated via a backflow transformation :

$$q_i = r_i + \sum_{j \neq i} \mathrm{Kernel}(r_{ij})(r_i - r_j)$$

It is also possible to compute the first and second derivatives

$$\nabla\phi_i(r_j) = \frac{d}{dx_j}\phi_i(r_j) + \frac{d}{dy_j}\phi_i(r_j) + \frac{d}{dz_j}\phi_i(r_j)$$

$$\mathrm{grad}\phi_i(r_j) = (\frac{d}{dx_j}\phi_i(r_j), \frac{d}{dy_j}\phi_i(r_j), \frac{d}{dz_j}\phi_i(r_j))$$

$$\Delta\phi_i(r_j) = \frac{d^2}{dx_j^2}\phi_i(r_j) + \frac{d^2}{dy_j^2}\phi_i(r_j) + \frac{d^2}{dz_j^2}\phi_i(r_j)$$

Parameters

- **pos** (*torch.tensor*) – Positions of the electrons Size : Nbatch, Nelec x Ndim
- **derivative** (*int, optional*) – order of the derivative (0,1,2,). Defaults to 0.
- **sum_grad** (*bool, optional*) – Return the sum_grad (i.e. the sum of the derivatives) or the individual terms. Defaults to True.
- **sum_hess** (*bool, optional*) – Return the sum_hess (i.e. the sum of the derivatives) or the individual terms. Defaults to True.
- **one_elec** (*bool, optional*) – if only one electron is in input

Returns

Value of the AO (or their derivatives)

size : Nbatch, Nelec, Norb (sum_grad = True)

size : Nbatch, Nelec, Norb, Ndim (sum_grad = False)

Return type torch.tensor

Examples::

```
>>> mol = Molecule('h2.xyz')
>>> ao = AtomicOrbitalsBackflow(mol)
>>> pos = torch.rand(100,6)
>>> aovals = ao(pos)
>>> daovals = ao(pos,derivative=1)
```

qmctorch.wavefunction.orbitals.atomic_orbitals_orbital_dependent_backflow module

class qmctorch.wavefunction.orbitals.atomic_orbitals_orbital_dependent_backflow.**AtomicOrbit**

Bases: *qmctorch.wavefunction.orbitals.atomic_orbitals.AtomicOrbitals*

Computes the value of atomic orbitals

Parameters

- **mol** (Molecule) – Molecule object
- **cuda** (*bool, optional*) – Turn GPU ON/OFF Defaults to False.

forward(*pos, derivative=[0], sum_grad=True, sum_hess=True, one_elec=False*)

Computes the values of the atomic orbitals.

$$\phi_i(r_j) = \sum_n c_n$$
$$textRad^i_n(r_j)$$
$$textY^i_n(r_j)$$

where Rad is the radial part and Y the spherical harmonics part. It is also possible to compute the first and second derivatives

$$nabla\phi_i(r_j) =$$
$$fracddx_j\phi_i(r_j)+$$
$$fracddy_j\phi_i(r_j)+$$
$$fracddz_j\phi_i(r_j)$$
$$textgrad\phi_i(r_j) = ($$
$$fracddx_j\phi_i(r_j),$$
$$fracddy_j\phi_i(r_j),$$
$$fracddz_j\phi_i(r_j))\Delta\phi_i(r_j) =$$
$$fracd^2dx_j^2\phi_i(r_j)+$$
$$fracd^2dy_j^2\phi_i(r_j)+$$
$$fracd^2dz_j^2\phi_i(r_j)$$

Parameters

- **pos** (*torch.tensor*) – Positions of the electrons Size : Nbatch, Nelec x Ndim
- **derivative** (*int, optional*) – order of the derivative (0,1,2,). Defaults to 0.
- **sum_grad** (*bool, optional*) – Return the sum_grad (i.e. the sum of the derivatives) or the individual terms. Defaults to True. False only for derivative=1
- **one_elec** (*bool, optional*) – if only one electron is in input

Returns

Value of the AO (or their derivatives) n size : Nbatch, Nelec, Norb (sum_grad = True) n size : Nbatch, Nelec, Norb, Ndim (sum_grad = False)

Return type torch.tensor

Examples::

```
>>> mol = Molecule('h2.xyz')
>>> ao = AtomicOrbitals(mol)
>>> pos = torch.rand(100,6)
>>> aovals = ao(pos)
>>> daovals = ao(pos,derivative=1)
```

qmctorch.wavefunction.orbitals.norm_orbital module

qmctorch.wavefunction.orbitals.norm_orbital.**atomic_orbital_norm**(*basis*)

Computes the norm of the atomic orbitals

Parameters **basis** (*Namespace*) – basis object of the Molecule instance

Returns Norm of the atomic orbitals

Return type torch.tensor

Examples::

```
>>> mol = Molecule('h2.xyz', basis='dzp', calculator='adf')
>>> norm = atomic_orbital_norm(mol.basis)
```

qmctorch.wavefunction.orbitals.norm_orbital.**norm_slater_spherical**(*bas_n*, *bas_exp*)

Normalization of STOs with Sphecrical Harmonics.

- www.theochem.ru.nl/~pwormer/Knowino/knowino.org/wiki/Slater_orbital
- C Filippi, JCP 105, 213 1996
- Monte Carlo Methods in Ab Inition Quantum Chemistry, B.L. Hammond

Parameters

- **bas_n** (*torch.tensor*) – prinicpal quantum number
- **bas_exp** (*torch.tensor*) – slater exponents

Returns normalization factor

Return type torch.tensor

qmctorch.wavefunction.orbitals.norm_orbital.**norm_gaussian_spherical**(*bas_n*, *bas_exp*)

Normlization of GTOs with spherical harmonics.

- Computational Quantum Chemistry: An interactive Intrduction to basis set theory

  eq : 1.14 page 23.

Parameters

- **bas_n** (*torch.tensor*) – prinicpal quantum number
- **bas_exp** (*torch.tensor*) – slater exponents

Returns normalization factor

Return type torch.tensor

qmctorch.wavefunction.orbitals.norm_orbital.**norm_slater_cartesian**(*a*, *b*, *c*, *n*, *exp*)

Normaliation of STos with cartesian harmonics.

- Monte Carlo Methods in Ab Initio Quantum Chemistry page 279

Parameters

- **a** (*torch.tensor*) – exponent of x
- **b** (*torch.tensor*) – exponent of y
- **c** (*torch.tensor*) – exponent of z
- **n** (*torch.tensor*) – exponent of r
- **exp** (*torch.tensor*) – Sater exponent

Returns normalization factor

Return type torch.tensor

qmctorch.wavefunction.orbitals.norm_orbital.**norm_gaussian_cartesian**(*a*, *b*, *c*, *exp*)

Normaliation of GTOs with cartesian harmonics.

- Monte Carlo Methods in Ab Initio Quantum Chemistry page 279

Parameters

- **a** (*torch.tensor*) – exponent of x
- **b** (*torch.tensor*) – exponent of y
- **c** (*torch.tensor*) – exponent of z
- **exp** (*torch.tensor*) – Sater exponent

Returns normalization factor

Return type torch.tensor

qmctorch.wavefunction.orbitals.radial_functions module

qmctorch.wavefunction.orbitals.radial_functions.**radial_slater**(*R*, *bas_n*, *bas_exp*, *xyz=None*, *derivative=0*, *sum_grad=True*, *sum_hess=True*)

Compute the radial part of STOs (or its derivative).

Parameters

- **R** (*torch.tensor*) – distance between each electron and each atom
- **bas_n** (*torch.tensor*) – principal quantum number
- **bas_exp** (*torch.tensor*) – exponents of the exponential

Keyword Arguments

- **xyz** (*torch.tensor*) – positions of the electrons (needed for derivative) (default: {None})
- **derivative** (*int*) – degree of the derivative (default: {0}) 0 : value of the function 1 : first derivative 2 : pure second derivative 3 : mixed second derivative
- **sum_grad** (*bool*) – return the sum_grad, i.e the sum of the gradients (default: {True})

- **sum_hess** (*bool*) – return the sum_hess, i.e the sum of the diag hessian (default: {False})
- **mixed_hess** (*bool*) –

  return the full hessian for each electron i.e. dxdy dxdz dydz ... mixed derivatives

  (default: {False})

Returns values of each orbital radial part at each position

Return type torch.tensor

qmctorch.wavefunction.orbitals.radial_functions.**radial_gaussian**(*R*, *bas_n*, *bas_exp*, *xyz=None*, *derivative=[0]*, *sum_grad=True*, *sum_hess=True*)

Compute the radial part of GTOs (or its derivative).

Parameters

- **R** (*torch.tensor*) – distance between each electron and each atom
- **bas_n** (*torch.tensor*) – principal quantum number
- **bas_exp** (*torch.tensor*) – exponents of the exponential

Keyword Arguments

- **xyz** (*torch.tensor*) – positions of the electrons (needed for derivative)(default: {None})
- **derivative** (*int*) – degree of the derivative(default: {0})
- **sum_grad** (*bool*) – return the sum_grad, i.e the sum of the gradients (default: {True})

Returns values of each orbital radial part at each position

Return type torch.tensor

qmctorch.wavefunction.orbitals.radial_functions.**radial_gaussian_pure**(*R*, *bas_n*, *bas_exp*, *xyz=None*, *derivative=[0]*, *sum_grad=True*, *sum_hess=True*)

Compute the radial part of GTOs (or its derivative).

Parameters

- **R** (*torch.tensor*) – distance between each electron and each atom
- **bas_n** (*torch.tensor*) – principal quantum number
- **bas_exp** (*torch.tensor*) – exponents of the exponential

Keyword Arguments

- **xyz** (*torch.tensor*) – positions of the electrons (needed for derivative)(default: {None})

- **derivative** (*int*) – degree of the derivative(default: {0})
- **sum_grad** (*bool*) – return the sum_grad, i.e the sum of the gradients (default: {True})
- **sum_hess** (*bool*) – return the sum_hess, i.e the sum of the lapacian (default: {True})

Returns values of each orbital radial part at each position

Return type torch.tensor

qmctorch.wavefunction.orbitals.radial_functions.**radial_slater_pure**(*R*, *bas_n*, *bas_exp*, *xyz=None*, *derivative=0*, *sum_grad=True*, *sum_hess=True*)

Compute the radial part of STOs (or its derivative).

Parameters

- **R** (*torch.tensor*) – distance between each electron and each atom
- **bas_n** (*torch.tensor*) – principal quantum number
- **bas_exp** (*torch.tensor*) – exponents of the exponential

Keyword Arguments

- **xyz** (*torch.tensor*) – positions of the electrons (needed for derivative)(default: {None})
- **derivative** (*int*) – degree of the derivative(default: {0})
- **sum_grad** (*bool*) – return the sum_grad, i.e the sum of the gradients (default: {True})
- **sum_hess** (*bool*) – return the sum_hess, i.e the sum of the laplacian (default: {True})

Returns values of each orbital radial part at each position

Return type torch.tensor

qmctorch.wavefunction.orbitals.radial_functions.**return_required_data**(*derivative*, *_kernel*, *_first_derivative_kernel*, *_second_derivative_kernel*, *_mixed_second_derivative_kern*

Returns the data contained in derivative

Parameters

- **derivative** (*list*) – list of the derivatives required
- **_kernel** (*callable*) – kernel of the values
- **_first_derivative_kernel** (*callable*) – kernel for 1st der
- **_second_derivative_kernel** (*callable*) – kernel for 2nd der

Returns values of the different der requried

Return type list

qmctorch.wavefunction.orbitals.spherical_harmonics module

class qmctorch.wavefunction.orbitals.spherical_harmonics.**Harmonics**(*type*, ***kwargs*)

Bases: object

Compute spherical or cartesian harmonics and their derivatives

Parameters **type** (*str*) – harmonics type (cart or sph)

Keyword Arguments

- **bas_l** (*torch.tensor*) – second quantum numbers (sph)
- **bas_m** (*torch.tensor*) – third quantum numbers (sph)
- **bas_kx** (*torch.tensor*) – x exponent (cart)
- **bas_ky** (*torch.tensor*) – xy exponent (cart)
- **bas_kz** (*torch.tensor*) – z exponent (cart)
- **cuda** (*bool*) – use cuda (defaults False)

Examples::

```
>>> mol = Molecule('h2.xyz')
>>> harm = Harmonics(cart)
>>> pos = torch.rand(100,6)
>>> hvals = harm(pos)
>>> dhvals = harm(pos,derivative=1)
```

qmctorch.wavefunction.orbitals.spherical_harmonics.**CartesianHarmonics**(*xyz, k, mask0, mask2, derivative=[0], sum_grad=True, sum_hess=True*)

Computes Real Cartesian Harmonics

$$Y = x^{k_x}$$
$$timesy^{k_y}$$
$$timesz^{k_z}$$

Parameters

- **xyz** (*torch.tensor*) – distance between sampling points and orbital centers n size : (Nbatch, Nelec, Nbas, Ndim)
- **k** (*torch.tensor*) – (kx,ky,kz) exponents
- **mask0** (*torch.tensor*) – precomputed mask of k=0
- **mask2** (*torch.tensor*) – precomputed mask of k=2
- **derivative** (*int, optional*) – degree of the derivative. Defaults to 0.
- **sum_grad** (*bool, optional*) – returns the sum of the derivative if True. Defaults to True.
- **sum_hess** (*bool, optional*) – returns the sum of the 2nd derivative if True. Defaults to True.

Returns values of the harmonics at the sampling points

Return type torch.tensor

qmctorch.wavefunction.orbitals.spherical_harmonics.**SphericalHarmonics**(*xyz*, *l*, *m*, *derivative=0*, *sum_grad=True*, *sum_hess=True*)

Compute the Real Spherical Harmonics of the AO.

Parameters

- **xyz** (*torch.tensor*) – distance between sampling points and orbital centers n size : (Nbatch, Nelec, Nbas, Ndim)
- **l** (*torch.tensor*) – l quantum number
- **m** (*torch.tensor*) – m quantum number

Returns

value of each harmonics at each points (or derivative) n size : (Nbatch,Nelec,Nrbf) for sum_grad=True n size : (Nbatch,Nelec,Nrbf, Ndim) for sum_grad=False

Return type Y (torch.tensor)

qmctorch.wavefunction.orbitals.spherical_harmonics.**get_spherical_harmonics**(*xyz*, *lval*, *m*, *derivative*)

Compute the Real Spherical Harmonics of the AO.

Parameters

- **xyz** (*torch.tensor*) – distance between sampling points and orbital centers n size : (Nbatch, Nelec, Nbas, Ndim)
- **l** (*torch.tensor*) – l quantum number
- **m** (*torch.tensor*) – m quantum number

Returns

value of each harmonics at each points (or derivative) n size : (Nbatch,Nelec,Nrbf)

Return type Y (torch.tensor)

qmctorch.wavefunction.orbitals.spherical_harmonics.**get_grad_spherical_harmonics**(*xyz*, *lval*, *m*)

Compute the gradient of the Real Spherical Harmonics of the AO.

Parameters

- **xyz** (*torch.tensor*) – distance between sampling points and orbital centers n size : (Nbatch, Nelec, Nbas, Ndim)
- **l** (*torch.tensor*) – l quantum number
- **m** (*torch.tensor*) – m quantum number

Returns

value of each harmonics at each points (or derivative) n size : (Nbatch,Nelec,Nrbf,3)

Return type Y (torch.tensor)

Module contents

qmctorch.wavefunction.pooling package

Submodules

qmctorch.wavefunction.pooling.orbital_configurations module

class qmctorch.wavefunction.pooling.orbital_configurations.**OrbitalConfigurations**(*mol*)
Bases: object

get_configs(*configs*)
Get the configuratio in the CI expansion

Parameters

- **configs** (*str*) – name of the configs we want
- **mol** (*mol object*) – molecule object

Returns the spin up/spin down electronic confs

Return type tuple(torch.LongTensor,torch.LongTensor)

sanity_check(*nelec*, *norb*)

Check if the number of elec/orb is consistent with the properties of the molecule

Parameters

- **nelec** (*int*) – required number of electrons in config
- **norb** (*int*) – required number of orb in config

qmctorch.wavefunction.pooling.orbital_configurations.**get_excitation**(*configs*)
get the excitation data

Parameters **configs** (*tuple*) – configuratin of the electrons

Returns

index of the obitals in the excitaitons [i,j],[l,m] : excitation i -> l, j -> l

Return type exc_up, exc_down

qmctorch.wavefunction.pooling.orbital_configurations.**get_unique_excitation**(*configs*)
get the unique excitation data

Parameters **configs** (*tuple*) – configuratin of the electrons

Returns

index of the obitals in the excitaitons [i,j],[l,m] : excitation i -> l, j -> l

index_up, index_down [index map for the unique exc] [0,0,...], [0,1,...] means that 1st : excitation is composed of unique_up[0]*unique_down[0] 2nd : excitation is composed of unique_up[0]*unique_down[1] ....

Return type exc_up, exc_down

qmctorch.wavefunction.pooling.orbital_projector module

class qmctorch.wavefunction.pooling.orbital_projector.**OrbitalProjector**(*configs*, *mol*, *cuda=False*)

Bases: object

Project the MO matrix in Slater Matrices

Parameters

- **configs** (*list*) – configurations of the slater determinants
- **mol** (Molecule) – Molecule object
- **cuda** (*bool*) – use cuda or not

get_projectors()

Get the projectors of the conf in the CI expansion

Returns projectors

Return type torch.tensor, torch.tensor

split_orbitals(*mat*)

Split the orbital matrix in multiple slater matrices

Parameters **mat** (*torch.tensor*) – matrix to split

Returns all slater matrices

Return type torch.tensor

class qmctorch.wavefunction.pooling.orbital_projector.**ExcitationMask**(*unique_excitations*, *mol*, *max_orb*, *cuda=False*)

Bases: object

Select the occupied MOs of Slater determinant using masks

Parameters

- **unique_excitations** (*list*) – the list of unique excitations
- **mol** (Molecule) – Molecule object
- **max_orb** (*list*) – the max index of each orb for each spin
- **cuda** (*bool*) – use cuda or not

get_index_unique_single()

Computes the 1D index and permutation for the unique singles.

get_index_unique_double()

Computes the 1D index of the double excitation matrices.

qmctorch.wavefunction.pooling.slater_pooling module

class qmctorch.wavefunction.pooling.slater_pooling.**SlaterPooling**(*config_method*, *configs*, *mol*, *cuda=False*)

Bases: sphinx.ext.autodoc.importer._MockObject

Computes the Sater determinants

Parameters

- **config_method** (*str*) – method used to define the config
- **configs** (*tuple*) – configuratin of the electrons
- **mol** (Molecule) – Molecule instance
- **cuda** (*bool, optional*) – Turns GPU ON/OFF. Defaults to False.

forward(*input*)
Computes the values of the determinats

Parameters **input** (*torch.tensor*) – MO matrices nbatch x nelec x nmo

Returns slater determinants

Return type torch.tensor

get_slater_matrices(*input*)
Computes the slater matrices

Parameters **input** (*torch.tensor*) – MO matrices nbatch x nelec x nmo

Returns slater matrices of spin up/down

Return type (torch.tensor, torch.tensor)

det_explicit(*input*)
Computes the values of the determinants from the slater matrices

Parameters **input** (*torch.tensor*) – MO matrices nbatch x nelec x nmo

Returns slater determinants

Return type torch.tensor

det_single_double(*input*)
Computes the determinant of ground state + single + double

Parameters **input** (*torch.tensor*) – MO matrices nbatch x nelec x nmo

Returns slater determinants

Return type torch.tensor

det_ground_state(*input*)
Computes the SD of the ground state

Parameters **input** (*torch.tensor*) – MO matrices nbatch x nelec x nmo

det_unique_single_double(*input*)
Computes the SD of single/double excitations

The determinants of the single excitations are calculated from the ground state determinant and the ground state Slater matrices whith one column modified. See : Monte Carlo Methods in ab initio quantum chemistry B.L. Hammond, appendix B1

Note : if the state on coonfigs are specified in order we end up with excitations that comes from a deep orbital, the resulting slater matrix has one column changed (with the new orbital) and several permutation. We therefore need to multiply the slater determinant by (-1)^nperm.

$$MO = [A|B]det(Exc_{ij}) = (det(A) * A^{-1} * B)_{i,j}$$

Parameters **input** (*torch.tensor*) – MO matrices nbatch x nelec x nmo

operator(*mo*, *bop*, *op=<built-in function add>*, *op_squared=False*)
Computes the values of an opearator applied to the procuts of determinant

Parameters

- **mo** (*torch.tensor*) – matrix of MO vals(Nbatch, Nelec, Nmo)
- **bkin** (*torch.tensor*) – kinetic operator (Nbatch, Nelec, Nmo)
- **op** (*operator*) – how to combine the up/down contribution
- **op_squared** (*bool, optional*) –

Returns kinetic energy

Return type torch.tensor

operator_ground_state(*mo*, *bop*, *op_squared=False*)
Computes the values of any operator on gs only

Parameters

- **mo** (*torch.tensor*) – matrix of molecular orbitals
- **bkin** (*torch.tensor*) – matrix of kinetic operator
- **op_squared** (*bool, optional*) –

Returns operator values

Return type torch.tensor

operator_explicit(*mo*, *bkin*, *op_squared=False*)

Computes the value of any operator using the trace trick for a product of spin up/down determinant.

$$-$$
$$frac12\Delta\Psi = -$$
$$frac12D_{up}D_{down}(\Delta_{up}D_{up}/D_{up} + \Delta_{down}D_{down}/D_{down})$$

Parameters

- **mo** (*torch.tensor*) – matrix of MO vals(Nbatch, Nelec, Nmo)
- **bkin** (*torch.tensor*) – kinetic operator (Nbatch, Nelec, Nmo)
- **op_squared** (*bool, optional*) –

Returns kinetic energy

Return type torch.tensor

operator_single_double(*mo*, *bop*, *op_squared=False*)
Computes the value of any operator on gs + single + double

Parameters

- **mo** (*torch.tensor*) – matrix of molecular orbitals
- **bkin** (*torch.tensor*) – matrix of kinetic operator
- **op_squared** (*bool, optional*) –

Returns kinetic energy values

Return type torch.tensor

operator_unique_single_double(*mo*, *bop*, *op_squared*)
Compute the operator value of the unique single/double conformation

Parameters

- **mo** (*[type]*) – [description]

- **bkin** ([type]) – [description]
- **op_squared** (bool) –

static op_single(*baseterm*, *mat_exc*, *M*, *index*, *nbatch*)
Computes the operator values for single excitation

$$Tr(\bar{A}^{-1}\bar{B}) = Tr(A^{-1}B) + Tr(TM)T = P(A^{-1}\bar{A})^{-1}PM = A^{-1}\bar{B} - A^{-1}BA^{-1}\bar{A}$$

Parameters

- **baseterm** (torch.tensor) – trace(A B)
- **mat_exc** (torch.tensor) – invA @ Abar
- **M** (torch.tensor) – invA Bbar - inv A B inv A Abar
- **index** (List) – list of index of the excitations
- **nbatch** – batch size

static op_multiexcitation(*baseterm*, *mat_exc*, *M*, *index*, *size*, *nbatch*)
Computes the operator values for single excitation

$$Tr(\bar{A}^{-1}\bar{B}) = Tr(A^{-1}B) + Tr(TM)T = P(A^{-1}\bar{A})^{-1}PM = A^{-1}\bar{B} - A^{-1}BA^{-1}\bar{A}$$

Parameters

- **baseterm** (torch.tensor) – trace(A B)
- **mat_exc** (torch.tensor) – invA @ Abar
- **M** (torch.tensor) – invA Bbar - inv A B inv A Abar
- **index** (List) – list of index of the excitations
- **size** (int) – number of excitation
- **nbatch** – batch size

static op_squared_single(*baseterm*, *mat_exc*, *M*, *Y*, *index*, *nbatch*)
Computes the operator squared for single excitation

$$Tr((\bar{A}^{-1}\bar{B})^2) = Tr((A^{-1}B)^2) + Tr((TM)^2) + 2Tr(TY)T = P(A^{-1}\bar{A})^{-1}P -> mat_excinthecodeM = A^{-1}\bar{B} - A$$

Parameters

- **baseterm** (torch.tensor) – trace(A B A B)
- **mat_exc** (torch.tensor) – invA @ Abar
- **M** (torch.tensor) – invA Bbar - inv A B inv A Abar
- **Y** (torch.tensor) – invA B M
- **index** (List) – list of index of the excitations
- **nbatch** – batch size

static op_squared_multiexcitation(*baseterm*, *mat_exc*, *M*, *Y*, *index*, *size*, *nbatch*)
Computes the operator squared for multiple excitation

$$Tr((\bar{A}^{-1}\bar{B})^2) = Tr((A^{-1}B)^2) + Tr((TM)^2) + 2Tr(TY)T = P(A^{-1}\bar{A})^{-1}P -> mat_excinthecodeM = A^{-1}\bar{B} - A$$

Parameters

- **baseterm** (*torch.tensor*) – trace(A B A B)
- **mat_exc** (*torch.tensor*) – invA @ Abar
- **M** (*torch.tensor*) – invA Bbar - inv A B inv A Abar
- **Y** (*torch.tensor*) – invA B M
- **index** (*List*) – list of index of the excitations
- **nbatch** – batch size
- **size** (*int*) – number of excitation

Module contents

14.5.2 Submodules

qmctorch.wavefunction.slater_combined_jastrow module

class qmctorch.wavefunction.slater_combined_jastrow.**SlaterCombinedJastrow**(*mol*, *configs='ground_state'*, *kinetic='jacobi'*, *jastrow_kernel={'ee': <class 'qmctorch.wavefunction.jastr 'een': None, 'en': <class 'qmctorch.wavefunction.jastr jastrow_kernel_kwargs={'e {}, 'een': {}, 'en': {}}, cuda=False, include_all_mo=True*)

Bases: *qmctorch.wavefunction.slater_jastrow.SlaterJastrow*

Slater Jastrow wave function with many body Jastrow factor

$$\Psi(R_{at}, r) = J(r) \sum_n c_n D_n^{\uparrow}(r^{\uparrow}) D_n^{\downarrow}(r^{\downarrow})$$

with

$$J(r) = \exp\left(K_{ee}(r) + K_{en}(R_{at}, r) + K_{een}(R_{at}, r)\right)$$

with the different kernels representing electron-electron, electron-nuclei and electron-electron-nuclei terms

Parameters

- **mol** (Molecule) – a QMCTorch molecule object
- **configs** (*str, optional*) – defines the CI configurations to be used. Defaults to 'ground_state'. - ground_state : only the ground state determinant in the wave function - single(n,m) : only single excitation with n electrons and m orbitals - single_double(n,m) : single and double excitation with n electrons and m orbitals - cas(n, m) : all possible configuration using n eletrons and m orbitals
- **kinetic** (*str, optional*) – method to compute the kinetic energy. Defaults to 'jacobi'. - jacobi : use the Jacobi formula to compute the kinetic energy - auto : use automatic differentiation to compute the kinetic energy
- **jastrow_kernel** (*dict, optional*) – different Jastrow kernels for the different terms. By default only electron-electron and electron-nuclei terms are used
- **jastrow_kernel_kwargs** (*dict, optional*) – keyword arguments for the jastrow kernels contructor
- **cuda** (*bool, optional*) – turns GPU ON/OFF Defaults to False.
- **include_all_mo** (*bool, optional*) – include either all molecular orbitals or only the ones that are popualted in the configs. Defaults to False

Examples::

```
>>> from qmctorch.scf import Molecule
>>> from qmctorch.wavefunction import SlaterCombinedJastrow
>>> mol = Molecule('h2o.xyz', calculator='adf', basis = 'dzp')
>>> wf = SlaterCombinedJastrow(mol, configs='cas(2,2)')
```

qmctorch.wavefunction.slater_jastrow module

class qmctorch.wavefunction.slater_jastrow.**SlaterJastrow**(*mol*, *configs='ground_state'*, *kinetic='jacobi'*, *jastrow_kernel=<class 'qmctorch.wavefunction.jastrows.elec_elec.kernels.pade_jastrow_kernel.PadeJastrowKernel'>*, *jastrow_kernel_kwargs={}*, *cuda=False*, *include_all_mo=True*)

Bases: *qmctorch.wavefunction.slater_jastrow_base.SlaterJastrowBase*

Slater Jastrow wave function with electron-electron Jastrow factor

$$\Psi(R_{at}, r) = J(r) \sum_n c_n D_n^{\uparrow}(r^{\uparrow}) D_n^{\downarrow}(r^{\downarrow})$$

with

$$J(r) = \exp\left(K_{ee}(r)\right)$$

with K, a kernel function depending only on the electron-eletron distances

Parameters

- **mol** (Molecule) – a QMCTorch molecule object

- **configs** (*str, optional*) – defines the CI configurations to be used. Defaults to 'ground_state'. - ground_state : only the ground state determinant in the wave function - single(n,m) : only single excitation with n electrons and m orbitals - single_double(n,m) : single and double excitation with n electrons and m orbitals - cas(n, m) : all possible configuration using n eletrons and m orbitals
- **kinetic** (*str, optional*) – method to compute the kinetic energy. Defaults to 'jacobi'. - jacobi : use the Jacobi formula to compute the kinetic energy - auto : use automatic differentiation to compute the kinetic energy
- **jastrow_kernel** (*JastrowKernelBase, optional*) – Class that computes the jastrow kernels
- **jastrow_kernel_kwargs** (*dict, optional*) – keyword arguments for the jastrow kernel contructor
- **cuda** (*bool, optional*) – turns GPU ON/OFF Defaults to Fals e.
- **include_all_mo** (*bool, optional*) – include either all molecular orbitals or only the ones that are popualted in the configs. Defaults to False

Examples::

```
>>> from qmctorch.scf import Molecule
>>> from qmctorch.wavefunction import SlaterJastrow
>>> mol = Molecule('h2o.xyz', calculator='adf', basis = 'dzp')
>>> wf = SlaterJastrow(mol, configs='cas(2,2)')
```

forward (*x*, *ao=None*)

computes the value of the wave function for the sampling points

$$\Psi(R) = J(R) \sum_n c_n D_n^u(r^u) \times D_n^d(r^d)$$

Parameters

- **x** (*torch.tensor*) – sampling points (Nbatch, 3*Nelec)
- **ao** (*torch.tensor, optional*) – values of the atomic orbitals (Nbatch, Nelec, Nao)

Returns values of the wave functions at each sampling point (Nbatch, 1)

Return type torch.tensor

Examples::

```
>>> mol = Molecule('h2.xyz', calculator='adf', basis = 'dzp')
>>> wf = SlaterJastrow(mol, configs='cas(2,2)')
>>> pos = torch.rand(500,6)
>>> vals = wf(pos)
```

ao2mo (*ao*)

Get the values of the MO from the values of AO.

pos2mo (*x*, *derivative=0*)

Get the values of MOs

Parameters **{torch.tensor} -- positions of the electrons [nbatch, nelec*ndim]** (*x*) –

Keyword Arguments **{int} -- order of the derivative (default** (*derivative*) – {0})

Returns torch.tensor – MO matrix [nbatch, nelec, nmo]

kinetic_energy_jacobi(*x*, ***kwargs*)

Compute the value of the kinetic enery using the Jacobi Formula. C. Filippi, Simple Formalism for Efficient Derivatives .

$$\frac{\Delta\Psi(R)}{\Psi(R)} = \Psi(R)^{-1}\sum_n c_n(\frac{\Delta D_n^u}{D_n^u} + \frac{\Delta D_n^d}{D_n^d})D_n^u D_n^d$$

We compute the laplacian of the determinants through the Jacobi formula

$$\frac{\Delta \det(A)}{\det(A)} = Tr(A^{-1}\Delta A)$$

Here $A = J(R)\phi$ and therefore :

$$\Delta A = (\Delta J)D + 2\nabla J \nabla D + (\Delta D)J$$

Parameters **x** (*torch.tensor*) – sampling points (Nbatch, 3*Nelec)

Returns values of the kinetic energy at each sampling points

Return type torch.tensor

gradients_jacobi(*x*, *sum_grad=False*, *pdf=False*)

Compute the gradients of the wave function (or density) using the Jacobi Formula C. Filippi, Simple Formalism for Efficient Derivatives.

$$\frac{K(R)}{\Psi(R)} = Tr(A^{-1}B_{grad})$$

The gradients of the wave function

$$\Psi(R) = J(R)\sum_n c_n D_n^u D_n^d = J(R)\Sigma$$

are computed following

$$\nabla\Psi(R) = (\nabla J(R))\,\Sigma + J(R)\,(\nabla\Sigma)$$

with

$$\nabla\Sigma = \sum_n c_n(\frac{\nabla D_n^u}{D_n^u} + \frac{\nabla D_n^d}{D_n^d})D_n^u D_n^d$$

that we compute with the Jacobi formula as:

$$\nabla\Sigma = \sum_n c_n(Tr((D_n^u)^{-1}\nabla D_n^u) + Tr((D_n^d)^{-1}\nabla D_n^d))D_n^u D_n^d$$

Parameters

- **x** (*torch.tensor*) – sampling points (Nbatch, 3*Nelec)
- **pdf** (*bool, optional*) – if true compute the grads of the density

Returns values of the gradients wrt the walker pos at each sampling points

Return type torch.tensor

get_kinetic_operator(*x*, *ao*, *dao*, *d2ao*, *mo*)
Compute the Bkin matrix

Parameters

- **x** (*torch.tensor*) – sampling points (Nbatch, 3*Nelec)
- **mo** (*torch.tensor, optional*) – precomputed values of the MOs

Returns matrix of the kinetic operator

Return type torch.tensor

qmctorch.wavefunction.slater_jastrow_backflow module

class qmctorch.wavefunction.slater_jastrow_backflow.**SlaterJastrowBackFlow**(*mol*, *configs='ground_state'*, *kinetic='jacobi'*, *jastrow_kernel=<class 'qmctorch.wavefunction.jastr*
jastrow_kernel_kwargs={}, *backflow_kernel=<class 'qmctorch.wavefunction.orbit*
backflow_kernel_kwargs={}, *orbital_dependent_backflo*
cuda=False, *include_all_mo=True*)

Bases: *qmctorch.wavefunction.slater_jastrow_base.SlaterJastrowBase*

Slater Jastrow wave function with electron-electron Jastrow factor and backflow

$$\Psi(R_{at}, r) = J(r) \sum_n c_n D_n^{\uparrow}(q^{\uparrow}) D_n^{\downarrow}(q^{\downarrow})$$

with

$$J(r) = \exp\left(K_{ee}(r)\right)$$

with K, a kernel function depending only on the electron-eletron distances, and

$$q(r_i) = r_i + \sum) j \neq i K_{BF}(r_{ij})(r_i - r_j)$$

is a backflow transformation defined by the kernel K_{BF}. Note that different transformation can be used for different orbital via the *orbital_dependent_backflow* option.

Args: :param mol: a QMCTorch molecule object :type mol: Molecule :param configs: defines the CI configurations to be used. Defaults to 'ground_state'.

- ground_state : only the ground state determinant in the wave function
- single(n,m) : only single excitation with n electrons and m orbitals
- single_double(n,m) : single and double excitation with n electrons and m orbitals
- cas(n, m) : all possible configuration using n eletrons and m orbitals

Parameters

- **kinetic** (*str, optional*) – method to compute the kinetic energy. Defaults to 'jacobi'. - jacobi : use the Jacobi formula to compute the kinetic energy - auto : use automatic differentiation to compute the kinetic energy
- **jastrow_kernel** (*JastrowKernelBase, optional*) – Class that computes the jastrow kernels
- **jastrow_kernel_kwargs** (*dict, optional*) – keyword arguments for the jastrow kernel contructor
- **backflow_kernel** (BackFlowKernelBase, *optional*) – kernel function of the backflow transformation. - By default an inverse kernel K(r_{ij}) = w/r_{ij} is used
- **backflow_kernel_kwargs** (*dict, optional*) – keyword arguments for the backflow kernel contructor
- **orbital_dependent_backflow** (*bool, optional*) – every orbital has a different transformation if True. Default to False
- **cuda** (*bool, optional*) – turns GPU ON/OFF Defaults to False.
- **include_all_mo** (*bool, optional*) – include either all molecular orbitals or only the ones that are popualted in the configs. Defaults to False

Examples::

```
>>> from qmctorch.scf import Molecule
>>> from qmctorch.wavefunction import SlaterJastrowBackFlow
>>> mol = Molecule('h2o.xyz', calculator='adf', basis = 'dzp')
>>> wf = SlaterJastrowBackFlow(mol, configs='cas(2,2)')
```

forward(*x*, *ao=None*)

computes the value of the wave function for the sampling points

$$J(R)\Psi(R) = J(R)\sum_n c_n D_n^u(r^u) \times D_n^d(r^d)$$

Parameters

- **x** (*torch.tensor*) – sampling points (Nbatch, 3*Nelec)
- **ao** (*torch.tensor, optional*) – values of the atomic orbitals (Nbatch, Nelec, Nao)

Returns values of the wave functions at each sampling point (Nbatch, 1)

Return type torch.tensor

Examples::

```
>>> mol = Molecule('h2.xyz', calculator='adf', basis = 'dzp')
>>> wf = SlaterJastrow(mol, configs='cas(2,2)')
>>> pos = torch.rand(500,6)
>>> vals = wf(pos)
```

ao2mo (*ao*)
transforms AO values in to MO values.

pos2mo (*x*, *derivative=0*, *sum_grad=True*)
Compute the MO vals from the pos

Parameters

- **x** ([type]) – [description]
- **derivative** (int, optional) – [description]. Defaults to 0.
- **sum_grad** (bool, optional) – [description]. Defaults to True.

Returns [description]

Return type [type]

kinetic_energy_jacobi (*x*, ***kwargs*)
Compute the value of the kinetic enery using the Jacobi Formula.

$$\frac{\Delta(J(R)\Psi(R))}{J(R)\Psi(R)} = \frac{\Delta J(R)}{J(R} + 2\frac{\nabla J(R)}{J(R)}\frac{\nabla\Psi(R)}{\Psi(R)} + \frac{\Delta\Psi(R)}{\Psi(R)}$$

The lapacian of the determinental part is computed via

$$\Delta_i\Psi(R)\sum_n c_n(\frac{\Delta_i D_n^u}{D_n^u} + \frac{\Delta_i D_n^d}{D_n^d} + 2\frac{\nabla_i D_n^u}{D_n^u}\frac{\nabla_i D_n^d}{D_n^d})D_n^u D_n^d$$

Since the backflow orbitals are multi-electronic the laplacian of the determinants are obtained

$$\frac{\Delta det(A)}{det(A)} = Tr(A^{-1}\Delta A) + Tr(A^{-1}\nabla A)Tr(A^{-1}\nabla A) + Tr((A^{-1}\nabla A)(A^{-1}\nabla A))$$

Parameters **x** (torch.tensor) – sampling points (Nbatch, 3*Nelec)

Returns values of the kinetic energy at each sampling points

Return type torch.tensor

gradients_jacobi (*x*, *sum_grad=True*)
Computes the gradients of the wf using Jacobi's Formula

Parameters **x** ([type]) – [description]

qmctorch.wavefunction.slater_jastrow_base module

class qmctorch.wavefunction.slater_jastrow_base.**SlaterJastrowBase**(*mol*, *configs='ground_state'*, *kinetic='jacobi'*, *cuda=False*, *include_all_mo=True*)

Bases: qmctorch.wavefunction.wf_base.WaveFunction

Implementation of the QMC Network.

Parameters

- **mol** (Molecule) – a QMCTorch molecule object

- **configs** (*str, optional*) – defines the CI configurations to be used. Defaults to 'ground_state'. - ground_state : only the ground state determinant in the wave function - single(n,m) : only single excitation with n electrons and m orbitals - single_double(n,m) : single and double excitation with n electrons and m orbitals - cas(n, m) : all possible configuration using n eletrons and m orbitals
- **kinetic** (*str, optional*) – method to compute the kinetic energy. Defaults to 'jacobi'. - jacobi : use the Jacobi formula to compute the kinetic energy - auto : use automatic differentiation to compute the kinetic energy
- **cuda** (*bool, optional*) – turns GPU ON/OFF Defaults to False.
- **include_all_mo** (*bool, optional*) – include either all molecular orbitals or only the ones that are popualted in the configs. Defaults to False

log_data()
Print information abut the wave function.

get_mo_coeffs()

update_mo_coeffs()

geometry(*pos*)
Returns the gemoetry of the system in xyz format

Parameters **pos** (*torch.tensor*) – sampling points (Nbatch, 3*Nelec)

Returns list where each element is one line of the xyz file

Return type list

gto2sto(*plot=False*)
Fits the AO GTO to AO STO. The SZ sto that have only one basis function per ao

forward(*x*, *ao=None*)
computes the value of the wave function for the sampling points

$$\Psi(R) = \sum_n c_n D_n^u(r^u) \times D_n^d(r^d)$$

Parameters

- **x** (*torch.tensor*) – sampling points (Nbatch, 3*Nelec)
- **ao** (*torch.tensor, optional*) – values of the atomic orbitals (Nbatch, Nelec, Nao)

Returns values of the wave functions at each sampling point (Nbatch, 1)

Return type torch.tensor

Examples::

```
>>> mol = Molecule('h2.xyz', calculator='adf', basis = 'dzp')
>>> wf = SlaterJastrow(mol, configs='cas(2,2)')
>>> pos = torch.rand(500,6)
>>> vals = wf(pos)
```

ao2mo(*ao*)
Get the values of the MO from the values of AO.

pos2mo(*x*, *derivative=0*)
Get the values of MOs from the positions

Parameters **{torch.tensor} -- positions of the electrons [nbatch, nelec*ndim]** (x) –

Keyword Arguments **{int} -- order of the derivative (default** (*derivative*) – {0})

Returns torch.tensor – MO matrix [nbatch, nelec, nmo]

kinetic_energy_jacobi (*x*, ***kwargs*)

Compute the value of the kinetic enery using the Jacobi Formula. C. Filippi, Simple Formalism for Efficient Derivatives .

$$\frac{K(R)}{\Psi(R)} = Tr(A^{-1}B_{kin})$$

Parameters **x** (*torch.tensor*) – sampling points (Nbatch, 3*Nelec)

Returns values of the kinetic energy at each sampling points

Return type torch.tensor

gradients_jacobi (*x*, *pdf=False*)

Compute the gradients of the wave function (or density) using the Jacobi Formula C. Filippi, Simple Formalism for Efficient Derivatives.

$$\frac{K(R)}{\Psi(R)} = Tr(A^{-1}B_{grad})$$

Parameters

- **x** (*torch.tensor*) – sampling points (Nbatch, 3*Nelec)
- **pdf** (*bool, optional*) – if true compute the grads of the density

Returns values of the gradients wrt the walker pos at each sampling points

Return type torch.tensor

get_gradient_operator (*x*, *ao*, *grad_ao*, *mo*)

Compute the gradient operator

Parameters

- **x** (*[type]*) – [description]
- **ao** (*[type]*) – [description]
- **dao** (*[type]*) – [description]

get_hessian_operator (*x*, *ao*, *dao*, *d2ao*, *mo*)

Compute the Bkin matrix

Parameters

- **x** (*torch.tensor*) – sampling points (Nbatch, 3*Nelec)
- **mo** (*torch.tensor, optional*) – precomputed values of the MOs

Returns matrix of the kinetic operator

Return type torch.tensor

qmctorch.wavefunction.slater_orbital_dependent_jastrow module

class qmctorch.wavefunction.slater_orbital_dependent_jastrow.**SlaterOrbitalDependentJastrow**

Bases: *qmctorch.wavefunction.slater_jastrow_base.SlaterJastrowBase*

Slater Jastrow Wave function with an orbital dependent Electron-Electron Jastrow Factor

$$\Psi(R_{at}, r) = \sum_n c_n D_n^{\uparrow}(r^{\uparrow}) D_n^{\downarrow}(r^{\downarrow})$$

where each molecular orbital of the determinants is multiplied with a different electron-electron Jastrow

$$\phi_i(r) \rightarrow J_i(r)\phi_i(r)$$

Parameters

- **mol** (*Molecule*) – a QMCTorch molecule object
- **configs** (*str, optional*) – defines the CI configurations to be used. Defaults to 'ground_state'. - ground_state : only the ground state determinant in the wave function - single(n,m) : only single excitation with n electrons and m orbitals - single_double(n,m) : single and double excitation with n electrons and m orbitals - cas(n, m) : all possible configuration using n eletrons and m orbitals
- **kinetic** (*str, optional*) – method to compute the kinetic energy. Defaults to 'jacobi'. - jacobi : use the Jacobi formula to compute the kinetic energy - auto : use automatic differentiation to compute the kinetic energy
- **jastrow_kernel** (*JastrowKernelBase, optional*) – Class that computes the jastrow kernels
- **jastrow_kernel_kwargs** (*dict, optional*) – keyword arguments for the jastrow kernel contructor
- **cuda** (*bool, optional*) – turns GPU ON/OFF Defaults to False.
- **include_all_mo** (*bool, optional*) – include either all molecular orbitals or only the ones that are popualted in the configs. Defaults to False

Examples::

```
>>> from qmctorch.scf import Molecule
>>> from qmctorch.wavefunction import SlaterOrbitalDependentJastrow
>>> mol = Molecule('h2o.xyz', calculator='adf', basis = 'dzp')
>>> wf = SlaterOrbitalDependentJastrow(mol, configs='cas(2,2)')
```

ordered_jastrow(*pos*, *derivative=0*, *sum_grad=True*)
Returns the value of the jastrow with the correct dimensions

Parameters

- **pos** (*torch.tensor*) – Positions of the electrons Size : Nbatch, Nelec x Ndim
- **derivative** (*int, optional*) – order of the derivative (0,1,2,). Defaults to 0.
- **sum_grad** (*bool, optional*) – Return the sum_grad (i.e. the sum of the derivatives) or the individual terms. Defaults to True. False only for derivative=1

Returns

value of the jastrow parameter for all confs Nbatch, Nelec, Nmo (sum_grad = True) Nbatch, Nelec, Nmo, Ndim (sum_grad = False)

Return type torch.tensor

forward(*x*, *ao=None*)
computes the value of the wave function for the sampling points

$$\Psi(R) = \sum_n c_n D_n^u(r^u) \times D_n^d(r^d)$$

Parameters

- **x** (*torch.tensor*) – sampling points (Nbatch, 3*Nelec)
- **ao** (*torch.tensor, optional*) – values of the atomic orbitals (Nbatch, Nelec, Nao)

Returns values of the wave functions at each sampling point (Nbatch, 1)

Return type torch.tensor

Examples::

```
>>> mol = Molecule('h2.xyz', calculator='adf', basis = 'dzp')
>>> wf = SlaterJastrow(mol, configs='cas(2,2)')
>>> pos = torch.rand(500,6)
>>> vals = wf(pos)
```

ao2mo(*ao*)
Get the values of the MO from the values of AO.

ao2cmo(*ao*, *jastrow*)

pos2mo(*x*, *derivative=0*, *sum_grad=True*)
Compute the uncorrelated MOs from the positions.

pos2cmo(*x*, *derivative=0*, *sum_grad=True*)
Get the values of correlated MOs

Parameters **{torch.tensor} -- positions of the electrons [nbatch, nelec*ndim]** (*x*) –

Returns torch.tensor – MO matrix [nbatch, nelec, nmo]

kinetic_energy_jacobi(*x*, ***kwargs*)
Compute the value of the kinetic enery using the Jacobi Formula. C. Filippi, Simple Formalism for

Efficient Derivatives .

$$\frac{K(R)}{\Psi(R)} = Tr(A^{-1}B_{kin})$$

Parameters **x** (*torch.tensor*) – sampling points (Nbatch, 3*Nelec)

Returns values of the kinetic energy at each sampling points

Return type torch.tensor

gradients_jacobi(*x*, *sum_grad=True*, *pdf=False*)
Computes the gradients of the wf using Jacobi's Formula

Parameters **x** (*[type]*) – [description]

get_hessian_operator(*x*)
Compute the Bkin matrix

Parameters

- **x** (*torch.tensor*) – sampling points (Nbatch, 3*Nelec)
- **mo** (*torch.tensor, optional*) – precomputed values of the MOs

Returns matrix of the kinetic operator

Return type torch.tensor

get_gradient_operator(*x*)
Compute the gradient operator

Parameters

- **x** (*[type]*) – [description]
- **ao** (*[type]*) – [description]
- **dao** (*[type]*) – [description]

qmctorch.wavefunction.wf_base module

class qmctorch.wavefunction.wf_base.**WaveFunction**(*nelec*, *ndim*, *kinetic='auto'*, *cuda=False*)
Bases: sphinx.ext.autodoc.importer._MockObject

forward(*x*)
Compute the value of the wave function. for a multiple conformation of the electrons

Parameters

- **parameters** – variational param of the wf
- **pos** – position of the electrons

Returns: values of psi

electronic_potential(*pos*)
Computes the electron-electron term

Parameters **x** (*torch.tensor*) – sampling points (Nbatch, 3*Nelec)

Returns values of the electon-electron energy at each sampling points

Return type torch.tensor

nuclear_potential (*pos*)
Computes the electron-nuclear term

Parameters **x** (*torch.tensor*) – sampling points (Nbatch, 3*Nelec)

Returns values of the electon-nuclear energy at each sampling points

Return type torch.tensor

nuclear_repulsion ()
Computes the nuclear-nuclear repulsion term

Returns values of the nuclear-nuclear energy at each sampling points

Return type torch.tensor

gradients_autograd (*pos*, *pdf=False*)
Computes the gradients of the wavefunction (or density) w.r.t the values of the pos.

Parameters

- **pos** (*torch.tensor*) – positions of the walkers
- **pdf** (*bool, optional*) – if true compute the grads of the density

Returns values of the gradients

Return type torch.tensor

kinetic_energy_autograd (*pos*)
Compute the kinetic energy through the 2nd derivative w.r.t the value of the pos.

Parameters **pos** (*torch.tensor*) – positions of the walkers

Returns values of nabla^2 * Psi

local_energy (*pos*)
Computes the local energy

$$E = K(R) + V_{ee}(R) + V_{en}(R) + V_{nn}$$

Parameters **pos** (*torch.tensor*) – sampling points (Nbatch, 3*Nelec)

Returns values of the local enrgies at each sampling points

Return type [torch.tensor]

Examples::

```
>>> mol = Molecule('h2.xyz', calculator='adf', basis = 'dzp')
>>> wf = SlaterJastrow(mol, configs='cas(2,2)')
>>> pos = torch.rand(500,6)
>>> vals = wf.local_energy(pos)
```

Note: by default kinetic_energy refers to kinetic_energy_autograd users can overwrite it to poit to any other methods see kinetic_energy_jacobi in wf_orbital

---

energy (*pos*)
Total energy for the sampling points.

variance (*pos*)
Variance of the energy at the sampling points.

sampling_error(*eloc*)
Compute the statistical uncertainty. Assuming the samples are uncorrelated.

pdf(*pos*, *return_grad=False*)
density of the wave function.

get_number_parameters()
Computes the total number of parameters.

load(*filename*, *group='wf_opt'*, *model='best'*)
Load trained parameters

Parameters

- **filename** (*str*) – hdf5 filename
- **group** (*str, optional*) – group in the hdf5 file where the model is stored. Defaults to 'wf_opt'.
- **model** (*str, optional*) – 'best' or ' last'. Defaults to 'best'.

14.5.3 Module contents

class qmctorch.wavefunction.**WaveFunction**(*nelec*, *ndim*, *kinetic='auto'*, *cuda=False*)
Bases: sphinx.ext.autodoc.importer._MockObject

forward(*x*)
Compute the value of the wave function. for a multiple conformation of the electrons

Parameters

- **parameters** – variational param of the wf
- **pos** – position of the electrons

Returns: values of psi

electronic_potential(*pos*)
Computes the electron-electron term

Parameters **x** (*torch.tensor*) – sampling points (Nbatch, 3*Nelec)

Returns values of the electon-electron energy at each sampling points

Return type torch.tensor

nuclear_potential(*pos*)
Computes the electron-nuclear term

Parameters **x** (*torch.tensor*) – sampling points (Nbatch, 3*Nelec)

Returns values of the electon-nuclear energy at each sampling points

Return type torch.tensor

nuclear_repulsion()
Computes the nuclear-nuclear repulsion term

Returns values of the nuclear-nuclear energy at each sampling points

Return type torch.tensor

gradients_autograd(*pos*, *pdf=False*)
Computes the gradients of the wavefunction (or density) w.r.t the values of the pos.

Parameters

- **pos** (*torch.tensor*) – positions of the walkers
- **pdf** (*bool, optional*) – if true compute the grads of the density

Returns values of the gradients

Return type torch.tensor

kinetic_energy_autograd(*pos*)
Compute the kinetic energy through the 2nd derivative w.r.t the value of the pos.

Parameters **pos** (*torch.tensor*) – positions of the walkers

Returns values of nabla^2 * Psi

local_energy(*pos*)
Computes the local energy

$$E = K(R) + V_{ee}(R) + V_{en}(R) + V_{nn}$$

Parameters **pos** (*torch.tensor*) – sampling points (Nbatch, 3*Nelec)

Returns values of the local enrgies at each sampling points

Return type [torch.tensor]

Examples::

```
>>> mol = Molecule('h2.xyz', calculator='adf', basis = 'dzp')
>>> wf = SlaterJastrow(mol, configs='cas(2,2)')
>>> pos = torch.rand(500,6)
>>> vals = wf.local_energy(pos)
```

Note: by default kinetic_energy refers to kinetic_energy_autograd users can overwrite it to poit to any other methods see kinetic_energy_jacobi in wf_orbital

---

energy(*pos*)
Total energy for the sampling points.

variance(*pos*)
Variance of the energy at the sampling points.

sampling_error(*eloc*)
Compute the statistical uncertainty. Assuming the samples are uncorrelated.

pdf(*pos*, *return_grad=False*)
density of the wave function.

get_number_parameters()
Computes the total number of parameters.

load(*filename*, *group='wf_opt'*, *model='best'*)
Load trained parameters

Parameters

- **filename** (*str*) – hdf5 filename
- **group** (*str, optional*) – group in the hdf5 file where the model is stored. Defaults to 'wf_opt'.
- **model** (*str, optional*) – 'best' or ' last'. Defaults to 'best'.

class qmctorch.wavefunction.**SlaterJastrow**(*mol*, *configs='ground_state'*, *kinetic='jacobi'*, *jastrow_kernel=<class 'qmctorch.wavefunction.jastrows.elec_elec.kernels.pade_jastrow_kernel.Pade*
jastrow_kernel_kwargs={}, *cuda=False*, *include_all_mo=True*)

Bases: *qmctorch.wavefunction.slater_jastrow_base.SlaterJastrowBase*

Slater Jastrow wave function with electron-electron Jastrow factor

$$\Psi(R_{at}, r) = J(r) \sum_n c_n D_n^{\uparrow}(r^{\uparrow}) D_n^{\downarrow}(r^{\downarrow})$$

with

$$J(r) = \exp\left(K_{ee}(r)\right)$$

with K, a kernel function depending only on the electron-eletron distances

Parameters

- **mol** (*Molecule*) – a QMCTorch molecule object
- **configs** (*str, optional*) – defines the CI configurations to be used. Defaults to 'ground_state'. - ground_state : only the ground state determinant in the wave function - single(n,m) : only single excitation with n electrons and m orbitals - single_double(n,m) : single and double excitation with n electrons and m orbitals - cas(n, m) : all possible configuration using n eletrons and m orbitals
- **kinetic** (*str, optional*) – method to compute the kinetic energy. Defaults to 'jacobi'. - jacobi : use the Jacobi formula to compute the kinetic energy - auto : use automatic differentiation to compute the kinetic energy
- **jastrow_kernel** (*JastrowKernelBase, optional*) – Class that computes the jastrow kernels
- **jastrow_kernel_kwargs** (*dict, optional*) – keyword arguments for the jastrow kernel contructor
- **cuda** (*bool, optional*) – turns GPU ON/OFF Defaults to Fals e.
- **include_all_mo** (*bool, optional*) – include either all molecular orbitals or only the ones that are popualted in the configs. Defaults to False

Examples::

```
>>> from qmctorch.scf import Molecule
>>> from qmctorch.wavefunction import SlaterJastrow
>>> mol = Molecule('h2o.xyz', calculator='adf', basis = 'dzp')
>>> wf = SlaterJastrow(mol, configs='cas(2,2)')
```

forward(*x*, *ao=None*)

computes the value of the wave function for the sampling points

$$\Psi(R) = J(R) \sum_n c_n D_n^u(r^u) \times D_n^d(r^d)$$

Parameters

- **x** (*torch.tensor*) – sampling points (Nbatch, 3*Nelec)
- **ao** (*torch.tensor, optional*) – values of the atomic orbitals (Nbatch, Nelec, Nao)

Returns values of the wave functions at each sampling point (Nbatch, 1)

Return type torch.tensor

Examples::

```
>>> mol = Molecule('h2.xyz', calculator='adf', basis = 'dzp')
>>> wf = SlaterJastrow(mol, configs='cas(2,2)')
>>> pos = torch.rand(500,6)
>>> vals = wf(pos)
```

ao2mo (*ao*)
Get the values of the MO from the values of AO.

pos2mo (*x*, *derivative=0*)
Get the values of MOs

Parameters **{torch.tensor} -- positions of the electrons [nbatch, nelec*ndim]** (*x*) –

Keyword Arguments **{int} -- order of the derivative (default** (*derivative*) – {0})

Returns torch.tensor – MO matrix [nbatch, nelec, nmo]

kinetic_energy_jacobi (*x*, ***kwargs*)
Compute the value of the kinetic enery using the Jacobi Formula. C. Filippi, Simple Formalism for Efficient Derivatives .

$$\frac{\Delta\Psi(R)}{\Psi(R)} = \Psi(R)^{-1}\sum_n c_n(\frac{\Delta D_n^u}{D_n^u} + \frac{\Delta D_n^d}{D_n^d})D_n^u D_n^d$$

We compute the laplacian of the determinants through the Jacobi formula

$$\frac{\Delta \det(A)}{\det(A)} = Tr(A^{-1}\Delta A)$$

Here $A = J(R)\phi$ and therefore :

$$\Delta A = (\Delta J)D + 2\nabla J \nabla D + (\Delta D)J$$

Parameters **x** (*torch.tensor*) – sampling points (Nbatch, 3*Nelec)

Returns values of the kinetic energy at each sampling points

Return type torch.tensor

gradients_jacobi (*x*, *sum_grad=False*, *pdf=False*)
Compute the gradients of the wave function (or density) using the Jacobi Formula C. Filippi, Simple Formalism for Efficient Derivatives.

$$\frac{K(R)}{\Psi(R)} = Tr(A^{-1}B_{grad})$$

The gradients of the wave function

$$\Psi(R) = J(R)\sum_n c_n D_n^u D_n^d = J(R)\Sigma$$

are computed following

$$\nabla\Psi(R) = (\nabla J(R))\,\Sigma + J(R)\,(\nabla\Sigma)$$

with

$$\nabla\Sigma = \sum_n c_n (\frac{\nabla D_n^u}{D_n^u} + \frac{\nabla D_n^d}{D_n^d}) D_n^u D_n^d$$

that we compute with the Jacobi formula as:

$$\nabla\Sigma = \sum_n c_n (Tr((D_n^u)^{-1}\nabla D_n^u) + Tr((D_n^d)^{-1}\nabla D_n^d)) D_n^u D_n^d$$

Parameters

- **x** (*torch.tensor*) – sampling points (Nbatch, 3*Nelec)
- **pdf** (*bool, optional*) – if true compute the grads of the density

Returns values of the gradients wrt the walker pos at each sampling points

Return type torch.tensor

get_kinetic_operator(*x*, *ao*, *dao*, *d2ao*, *mo*)

Compute the Bkin matrix

Parameters

- **x** (*torch.tensor*) – sampling points (Nbatch, 3*Nelec)
- **mo** (*torch.tensor, optional*) – precomputed values of the MOs

Returns matrix of the kinetic operator

Return type torch.tensor

class qmctorch.wavefunction.**SlaterCombinedJastrow**(*mol*, *configs='ground_state'*, *kinetic='jacobi'*, *jastrow_kernel={'ee': <class 'qmctorch.wavefunction.jastrows.elec_elec.kernels.pade_jastrow_ 'een': None, 'en': <class 'qmctorch.wavefunction.jastrows.elec_nuclei.kernels.pade_jastro jastrow_kernel_kwargs={'ee': {}, 'een': {}, 'en': {}}*, *cuda=False*, *include_all_mo=True*)

Bases: *qmctorch.wavefunction.slater_jastrow.SlaterJastrow*

Slater Jastrow wave function with many body Jastrow factor

$$\Psi(R_{at}, r) = J(r) \sum_n c_n D_n^{\uparrow}(r^{\uparrow}) D_n^{\downarrow}(r^{\downarrow})$$

with

$$J(r) = \exp\left(K_{ee}(r) + K_{en}(R_{at}, r) + K_{een}(R_{at}, r)\right)$$

with the different kernels representing electron-electron, electron-nuclei and electron-electron-nuclei terms

Parameters

- **mol** (Molecule) – a QMCTorch molecule object
- **configs** (*str, optional*) – defines the CI configurations to be used. Defaults to 'ground_state'. - ground_state : only the ground state determinant in the wave function - single(n,m) : only single excitation with n electrons and m orbitals - single_double(n,m) : single and double excitation with n electrons and m orbitals - cas(n, m) : all possible configuration using n eletrons and m orbitals

- **kinetic** (*str, optional*) – method to compute the kinetic energy. Defaults to 'jacobi'. - jacobi : use the Jacobi formula to compute the kinetic energy - auto : use automatic differentiation to compute the kinetic energy
- **jastrow_kernel** (*dict, optional*) – different Jastrow kernels for the different terms. By default only electron-electron and electron-nuclei terms are used
- **jastrow_kernel_kwargs** (*dict, optional*) – keyword arguments for the jastrow kernels contructor
- **cuda** (*bool, optional*) – turns GPU ON/OFF Defaults to False.
- **include_all_mo** (*bool, optional*) – include either all molecular orbitals or only the ones that are popualted in the configs. Defaults to False

Examples::

```
>>> from qmctorch.scf import Molecule
>>> from qmctorch.wavefunction import SlaterCombinedJastrow
>>> mol = Molecule('h2o.xyz', calculator='adf', basis = 'dzp')
>>> wf = SlaterCombinedJastrow(mol, configs='cas(2,2)')
```

class qmctorch.wavefunction.**SlaterJastrowBackFlow**(*mol*, *configs='ground_state'*, *kinetic='jacobi'*, *jastrow_kernel=<class 'qmctorch.wavefunction.jastrows.elec_elec.kernels.pade_jastrow_*, *jastrow_kernel_kwargs={}*, *backflow_kernel=<class 'qmctorch.wavefunction.orbitals.backflow.kernels.backflow_kerne*, *backflow_kernel_kwargs={}*, *orbital_dependent_backflow=False*, *cuda=False*, *include_all_mo=True*)

Bases: *qmctorch.wavefunction.slater_jastrow_base.SlaterJastrowBase*

Slater Jastrow wave function with electron-electron Jastrow factor and backflow

$$\Psi(R_{at}, r) = J(r) \sum_n c_n D_n^{\uparrow}(q^{\uparrow}) D_n^{\downarrow}(q^{\downarrow})$$

with

$$J(r) = \exp\left(K_{ee}(r)\right)$$

with K, a kernel function depending only on the electron-eletron distances, and

$$q(r_i) = r_i + \sum) j \neq i K_{BF}(r_{ij})(r_i - r_j)$$

is a backflow transformation defined by the kernel K_{BF}. Note that different transformation can be used for different orbital via the *orbital_dependent_backflow* option.

Args: :param mol: a QMCTorch molecule object :type mol: Molecule :param configs: defines the CI configurations to be used. Defaults to 'ground_state'.

- ground_state : only the ground state determinant in the wave function
- single(n,m) : only single excitation with n electrons and m orbitals
- single_double(n,m) : single and double excitation with n electrons and m orbitals
- cas(n, m) : all possible configuration using n eletrons and m orbitals

Parameters

- **kinetic** (*str, optional*) – method to compute the kinetic energy. Defaults to 'jacobi'. - jacobi : use the Jacobi formula to compute the kinetic energy - auto : use automatic differentiation to compute the kinetic energy
- **jastrow_kernel** (*JastrowKernelBase, optional*) – Class that computes the jastrow kernels
- **jastrow_kernel_kwargs** (*dict, optional*) – keyword arguments for the jastrow kernel contructor
- **backflow_kernel** (BackFlowKernelBase, *optional*) – kernel function of the backflow transformation. - By default an inverse kernel K(r_{ij}) = w/r_{ij} is used
- **backflow_kernel_kwargs** (*dict, optional*) – keyword arguments for the backflow kernel contructor
- **orbital_dependent_backflow** (*bool, optional*) – every orbital has a different transformation if True. Default to False
- **cuda** (*bool, optional*) – turns GPU ON/OFF Defaults to False.
- **include_all_mo** (*bool, optional*) – include either all molecular orbitals or only the ones that are popualted in the configs. Defaults to False

Examples::

```
>>> from qmctorch.scf import Molecule
>>> from qmctorch.wavefunction import SlaterJastrowBackFlow
>>> mol = Molecule('h2o.xyz', calculator='adf', basis = 'dzp')
>>> wf = SlaterJastrowBackFlow(mol, configs='cas(2,2)')
```

forward (*x*, *ao=None*)

computes the value of the wave function for the sampling points

$$J(R)\Psi(R) = J(R)\sum_n c_n D_n^u(r^u) \times D_n^d(r^d)$$

Parameters

- **x** (*torch.tensor*) – sampling points (Nbatch, 3*Nelec)
- **ao** (*torch.tensor, optional*) – values of the atomic orbitals (Nbatch, Nelec, Nao)

Returns values of the wave functions at each sampling point (Nbatch, 1)

Return type torch.tensor

Examples::

```
>>> mol = Molecule('h2.xyz', calculator='adf', basis = 'dzp')
>>> wf = SlaterJastrow(mol, configs='cas(2,2)')
>>> pos = torch.rand(500,6)
>>> vals = wf(pos)
```

ao2mo (*ao*)

transforms AO values in to MO values.

pos2mo (*x*, *derivative=0*, *sum_grad=True*)

Compute the MO vals from the pos

Parameters

- **x** ([type]) – [description]
- **derivative** (int, optional) – [description]. Defaults to 0.
- **sum_grad** (bool, optional) – [description]. Defaults to True.

Returns [description]

Return type [type]

kinetic_energy_jacobi(*x*, ***kwargs*)

Compute the value of the kinetic enery using the Jacobi Formula.

$$\frac{\Delta(J(R)\Psi(R))}{J(R)\Psi(R)} = \frac{\Delta J(R)}{J(R} + 2\frac{\nabla J(R)}{J(R)}\frac{\nabla\Psi(R)}{\Psi(R)} + \frac{\Delta\Psi(R)}{\Psi(R)}$$

The lapacian of the determinental part is computed via

$$\Delta_i\Psi(R)\sum_n c_n(\frac{\Delta_i D_n^u}{D_n^u} + \frac{\Delta_i D_n^d}{D_n^d} + 2\frac{\nabla_i D_n^u}{D_n^u}\frac{\nabla_i D_n^d}{D_n^d})D_n^u D_n^d$$

Since the backflow orbitals are multi-electronic the laplacian of the determinants are obtained

$$\frac{\Delta det(A)}{det(A)} = Tr(A^{-1}\Delta A) + Tr(A^{-1}\nabla A)Tr(A^{-1}\nabla A) + Tr((A^{-1}\nabla A)(A^{-1}\nabla A))$$

Parameters **x** (torch.tensor) – sampling points (Nbatch, 3*Nelec)

Returns values of the kinetic energy at each sampling points

Return type torch.tensor

gradients_jacobi(*x*, *sum_grad=True*)

Computes the gradients of the wf using Jacobi's Formula

Parameters **x** ([type]) – [description]

class qmctorch.wavefunction.**SlaterOrbitalDependentJastrow**(*mol*, *configs='ground_state'*, *kinetic='jacobi'*, *jastrow_kernel=<class 'qmctorch.wavefunction.jastrows.elec_elec.kernels.pa*
jastrow_kernel_kwargs={}, *cuda=False*, *include_all_mo=True*)

Bases: qmctorch.wavefunction.slater_jastrow_base.SlaterJastrowBase

Slater Jastrow Wave function with an orbital dependent Electron-Electron Jastrow Factor

$$\Psi(R_{at}, r) = \sum_n c_n D_n^{\uparrow}(r^{\uparrow})D_n^{\downarrow}(r^{\downarrow})$$

where each molecular orbital of the determinants is multiplied with a different electron-electron Jastrow

$$\phi_i(r) \rightarrow J_i(r)\phi_i(r)$$

Parameters

- **mol** (Molecule) – a QMCTorch molecule object

- **configs** (*str, optional*) – defines the CI configurations to be used. Defaults to 'ground_state'. - ground_state : only the ground state determinant in the wave function - single(n,m) : only single excitation with n electrons and m orbitals - single_double(n,m) : single and double excitation with n electrons and m orbitals - cas(n, m) : all possible configuration using n eletrons and m orbitals
- **kinetic** (*str, optional*) – method to compute the kinetic energy. Defaults to 'jacobi'. - jacobi : use the Jacobi formula to compute the kinetic energy - auto : use automatic differentiation to compute the kinetic energy
- **jastrow_kernel** (*JastrowKernelBase, optional*) – Class that computes the jastrow kernels
- **jastrow_kernel_kwargs** (*dict, optional*) – keyword arguments for the jastrow kernel contructor
- **cuda** (*bool, optional*) – turns GPU ON/OFF Defaults to False.
- **include_all_mo** (*bool, optional*) – include either all molecular orbitals or only the ones that are popualted in the configs. Defaults to False

Examples::

```
>>> from qmctorch.scf import Molecule
>>> from qmctorch.wavefunction import SlaterOrbitalDependentJastrow
>>> mol = Molecule('h2o.xyz', calculator='adf', basis = 'dzp')
>>> wf = SlaterOrbitalDependentJastrow(mol, configs='cas(2,2)')
```

ordered_jastrow(*pos*, *derivative=0*, *sum_grad=True*)

Returns the value of the jastrow with the correct dimensions

Parameters

- **pos** (*torch.tensor*) – Positions of the electrons Size : Nbatch, Nelec x Ndim
- **derivative** (*int, optional*) – order of the derivative (0,1,2,). Defaults to 0.
- **sum_grad** (*bool, optional*) – Return the sum_grad (i.e. the sum of the derivatives) or the individual terms. Defaults to True. False only for derivative=1

Returns

value of the jastrow parameter for all confs Nbatch, Nelec, Nmo (sum_grad = True) Nbatch, Nelec, Nmo, Ndim (sum_grad = False)

Return type torch.tensor

forward(*x*, *ao=None*)

computes the value of the wave function for the sampling points

$$\Psi(R) = \sum_n c_n D_n^u(r^u) \times D_n^d(r^d)$$

Parameters

- **x** (*torch.tensor*) – sampling points (Nbatch, 3*Nelec)
- **ao** (*torch.tensor, optional*) – values of the atomic orbitals (Nbatch, Nelec, Nao)

Returns values of the wave functions at each sampling point (Nbatch, 1)

Return type torch.tensor

Examples::

```
>>> mol = Molecule('h2.xyz', calculator='adf', basis = 'dzp')
>>> wf = SlaterJastrow(mol, configs='cas(2,2)')
>>> pos = torch.rand(500,6)
>>> vals = wf(pos)
```

ao2mo (*ao*)
Get the values of the MO from the values of AO.

ao2cmo (*ao*, *jastrow*)

pos2mo (*x*, *derivative=0*, *sum_grad=True*)
Compute the uncorrelated MOs from the positions.

pos2cmo (*x*, *derivative=0*, *sum_grad=True*)
Get the values of correlated MOs

Parameters **{torch.tensor} -- positions of the electrons [nbatch, nelec*ndim]** (*x*) –

Returns torch.tensor – MO matrix [nbatch, nelec, nmo]

kinetic_energy_jacobi (*x*, ***kwargs*)
Compute the value of the kinetic enery using the Jacobi Formula. C. Filippi, Simple Formalism for Efficient Derivatives .

$$\frac{K(R)}{\Psi(R)} = Tr(A^{-1}B_{kin})$$

Parameters **x** (*torch.tensor*) – sampling points (Nbatch, 3*Nelec)

Returns values of the kinetic energy at each sampling points

Return type torch.tensor

gradients_jacobi (*x*, *sum_grad=True*, *pdf=False*)
Computes the gradients of the wf using Jacobi's Formula

Parameters **x** (*[type]*) – [description]

get_hessian_operator (*x*)
Compute the Bkin matrix

Parameters

- **x** (*torch.tensor*) – sampling points (Nbatch, 3*Nelec)
- **mo** (*torch.tensor, optional*) – precomputed values of the MOs

Returns matrix of the kinetic operator

Return type torch.tensor

get_gradient_operator (*x*)
Compute the gradient operator

Parameters

- **x** (*[type]*) – [description]
- **ao** (*[type]*) – [description]
- **dao** (*[type]*) – [description]

class qmctorch.wavefunction.**SlaterCombinedJastrowBackflow**(*mol*, *configs='ground_state'*, *kinetic='jacobi'*, *jastrow_kernel={'ee': <class 'qmctorch.wavefunction.jastrows.elec_elec.kernels.pa 'een': None*, *'en': <class 'qmctorch.wavefunction.jastrows.elec_nuclei.kernels.* *jastrow_kernel_kwargs={'ee': {}*, *'een': {}*, *'en': {}}*, *backflow_kernel=<class 'qmctorch.wavefunction.orbitals.backflow.kernels.bac* *backflow_kernel_kwargs={}*, *orbital_dependent_backflow=False*, *cuda=False*, *include_all_mo=True*)

Bases: *qmctorch.wavefunction.slater_jastrow.SlaterJastrow*

Slater Jastrow wave function with many-body Jastrow factor and backflow

$$\Psi(R_{at}, r) = J(R_{at}, r) \sum_n c_n D_n^{\uparrow}(q^{\uparrow}) D_n^{\downarrow}(q^{\downarrow})$$

with

$$J(r) = \exp\left(K_{ee}(r) + K_{en}(R_{at}, r) + K_{een}(R_{at}, r)\right)$$

with the different kernels representing electron-electron, electron-nuclei and electron-electron-nuclei terms and

$$q(r_i) = r_i + \sum) j \neq i K_{BF}(r_{ij})(r_i - r_j)$$

is a backflow transformation defined by the kernel K_{BF}. Note that different transformation can be used for different orbital via the *orbital_dependent_backflow* option.

Args: :param mol: a QMCTorch molecule object :type mol: Molecule :param configs: defines the CI configurations to be used. Defaults to 'ground_state'.

- ground_state : only the ground state determinant in the wave function
- single(n,m) : only single excitation with n electrons and m orbitals
- single_double(n,m) : single and double excitation with n electrons and m orbitals
- cas(n, m) : all possible configuration using n eletrons and m orbitals

Parameters

- **kinetic** (*str, optional*) – method to compute the kinetic energy. Defaults to 'jacobi'. - jacobi : use the Jacobi formula to compute the kinetic energy - auto : use automatic differentiation to compute the kinetic energy
- **jastrow_kernel** (*dict, optional*) – different Jastrow kernels for the different terms. By default only electron-electron and electron-nuclei terms are used
- **jastrow_kernel_kwargs** (*dict, optional*) – keyword arguments for the jastrow kernels contructor

- **backflow_kernel** (*BackFlowKernelBase, optional*) – kernel function of the backflow transformation. - By default an inverse kernel K(r_{ij}) = w/r_{ij} is used
- **backflow_kernel_kwargs** (*dict, optional*) – keyword arguments for the backflow kernel contructor
- **orbital_dependent_backflow** (*bool, optional*) – every orbital has a different transformation if True. Default to False
- **cuda** (*bool, optional*) – turns GPU ON/OFF Defaults to False.
- **include_all_mo** (*bool, optional*) – include either all molecular orbitals or only the ones that are popualted in the configs. Defaults to False

Examples::

```
>>> from qmctorch.scf import Molecule
>>> from qmctorch.wavefunction import SlaterCombinedJastrowBackflow
>>> mol = Molecule('h2o.xyz', calculator='adf', basis = 'dzp')
>>> wf = SlaterCombinedJastrowBackflow(mol, configs='cas(2,2)')
```

forward (*x*, *ao=None*)

computes the value of the wave function for the sampling points

$$J(R)\Psi(R) = J(R)\sum_n c_n D_n^u(r^u) \times D_n^d(r^d)$$

Parameters

- **x** (*torch.tensor*) – sampling points (Nbatch, 3*Nelec)
- **ao** (*torch.tensor, optional*) – values of the atomic orbitals (Nbatch, Nelec, Nao)

Returns values of the wave functions at each sampling point (Nbatch, 1)

Return type torch.tensor

Examples::

```
>>> mol = Molecule('h2.xyz', calculator='adf', basis = 'dzp')
>>> wf = SlaterJastrow(mol, configs='cas(2,2)')
>>> pos = torch.rand(500,6)
>>> vals = wf(pos)
```

ao2mo (*ao*)

transforms AO values in to MO values.

pos2mo (*x*, *derivative=0*, *sum_grad=True*)

Compute the MO vals from the pos

Parameters

- **x** (*[type]*) – [description]
- **derivative** (*int, optional*) – [description]. Defaults to 0.
- **sum_grad** (*bool, optional*) – [description]. Defaults to True.

Returns [description]

Return type [type]

kinetic_energy_jacobi(*x*, ***kwargs*)

Compute the value of the kinetic enery using the Jacobi Formula.

$$
\begin{aligned}
frac\Delta(J(R)\Psi(R))J(R)\Psi(R) = \\
frac \\
DeltaJ(R)J(R+2 \\
frac \\
nablaJ(R)J(R) \\
frac \\
nabla \\
Psi(R) \\
Psi(R) + \\
frac \\
Delta \\
Psi(R) \\
Psi(R)
\end{aligned}
$$

The lapacian of the determinental part is computed via

$$
\begin{aligned}
Delta_i \\
Psi(R) \\
sum_n c_n(\\
frac \\
Delta_i D_n^u D_n^u + \\
frac \\
Delta_i D_n^d D_n^d + 2 \\
frac \\
nabla_i D_n^u D_n^u \\
frac \\
nabla_i D_n^d D_n^d) D_n^u D_n^d
\end{aligned}
$$

Since the backflow orbitals are multi-electronic the laplacian of the determinants are obtained

$$
\begin{aligned}
frac \\
Deltadet(A)det(A) = Tr(A^{-1} \\
DeltaA) + Tr(A^{-1} \\
nablaA)Tr(A^{-1} \\
nablaA) + Tr((A^{-1} \\
nablaA)(A^{-1} \\
nablaA))
\end{aligned}
$$

Parameters **x** (*torch.tensor*) – sampling points (Nbatch, 3*Nelec)

Returns values of the kinetic energy at each sampling points

Return type torch.tensor

gradients_jacobi(*x*, *sum_grad=True*)
Computes the gradients of the wf using Jacobi's Formula

Parameters **x** ([type]) – [description]

CHAPTER 15

Module contents

Documentation about QMCTorch

CHAPTER 16

Indices and tables

- genindex
- modindex
- search

Python Module Index

q

Index

C

D

E

F

G

J

K

L

M

N

O

P

Q

R

S

T

U

V

W