moha Documentation

Release 1.0.0

Yilin Zhao

Aug 21, 2018

CONTENTS

MoHa is abbreviation of *Mo*lecular/*Mo*del *Ha*miltonian, it is a quantum chemistry program written in python. MoHa now is not a formal program but a toy for practice coding and help myself to have a better understanding of quantum chemistry. If you are interested in, Please see more details in the Overview.

CHAPTER

ONE

CONTENTS

1.1 MoHa Overview

Python-based simulations of chemistry framework (PYSCF) is a general-purpose electronic structure platform designed from the ground up to emphasize code simplicity, so as to facilitate new method development and enable flexible computational workflows. The package provides a wide range of tools to support simulations of finite-size systems, extended systems with periodic boundary conditions, low-dimensional periodic systems, and custom Hamiltonians, using mean-field and post-mean-field methods with standard Gaussian basis functions. To ensure ease of extensibility, PYSCF uses the Python language to implement almost all of its features, while computationally critical paths are implemented with heavily optimized C routines. Using this combined Python/C implementation, the package is as efficient as the best existing C or Fortran- based quantum chemistry programs.

1.1.1 Features

- Interface to integral package Libcint
- Interface to DMRG CheMPS2
- Interface to DMRG Block
- Interface to FCIQMC NECI
- Interface to XC functional library XCFun
- Interface to XC functional library Libxc

1.2 Installation

1.3 Molecular System

To begin a calculation with MoHa, the first step is to build a Hamiltonian of a system, either molecular system or model system. In most cases, we need to build a molecular system

In terms of second quantisation operators, a general Hamiltonian can be written as

$$H = -\sum_{ij} t_{ij} \hat{c}_i^\dagger \hat{c}_j + \frac{1}{2} \sum_{ijkl} V_{ijkl} \hat{c}_i^\dagger \hat{c}_k^\dagger \hat{c}_l \hat{c}_j$$

The construction of molecular Hamiltonian usually set up in three steps.

- First, construct a molecular geometry.

- Second, generate a Gaussian basis set for the molecular.
- Finally, compute all kinds of one body terms and two body terms with that basis to define a Hamiltonian.

1.3.1 Molecule

Molecule is a system consist with nucleus and electrons. For quantum chemistry calculation, we will always used the Born-Oppenheimer apporimation, which assumption that the motion of atomic nuclei and electrons in a molecule can be separated

$$\Psi_{molecule} = \psi_{electronic} \otimes \psi_{nuclear}$$

The module `molecule` in MoHa actually only contains imformation of the nuclear. It has three class:

- `Element`

Represents an element from the periodic table. The following attributes are supported for all elements:

number The atomic number.

symbol A string with the symbol of the element.

name The full element name.

group The group of the element (not for actinides and lanthanides).

period The row of the periodic system.

- `Atom`

Represents an Atom. The following attributes are supported for all atoms:

element A object of `Element` class.

coordinate The coordinate of the atom object.

- `Molecule`

Represents an Molecule. The following attributes are supported for all molecule object:

title type: string title of the system

size type: intager number of atoms

symmetry type: string point group of the molecule

bond_length

Calculate the interatomic distances using the expression:

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

where x, y, and z are Cartesian coordinates and i and j denote atomic indices.

bond_angle Calculate all possible bond angles. For example, the angle, ϕ_{ijk}, between atoms i-j-k, where j is the central atom is given by:

$$\cos\phi_{ijk} = \mathbf{\tilde{e}_{ji}} \cdot \mathbf{\tilde{e}_{jk}}$$

where the $\mathbf{\tilde{e}_{ij}}$ are unit vectors between the atoms, e.g.,

$$e_{ij}^{x} = -\left(x_i - x_j\right)/R_{ij}, \quad e_{ij}^{y} = -\left(y_i - y_j\right)/R_{ij}, \quad e_{ij}^{z} = -\left(z_i - z_j\right)/R_{ij}$$

out_of_plane_angle Calculate all possible out-of-plane angles. For example, the angle θ_{ijkl} for atom i out of the plane containing atoms j-k-l (with k as the central atom, connected to i) is given by:

$$\sin\theta_{ijkl} = \frac{\tilde{\mathbf{e}}_{\mathbf{kj}} \times \tilde{\mathbf{e}}_{\mathbf{kl}}}{\sin\phi_{jkl}} \cdot \tilde{\mathbf{e}}_{\mathbf{ki}}$$

dihedral_angle

Calculate all possible torsional angles. For example, the torsional angle τ_{ijkl} for the atom connectivity i-j-k-l is given by:

$$\cos\tau_{ijkl} = \frac{\left(\tilde{\mathbf{e}}_{\mathbf{ij}} \times \tilde{\mathbf{e}}_{\mathbf{jk}}\right) \cdot \left(\tilde{\mathbf{e}}_{\mathbf{jk}} \times \tilde{\mathbf{e}}_{\mathbf{kl}}\right)}{\sin\phi_{ijk} \quad \sin\phi_{jkl}}$$

To be convenient, we can specify the molecular object by load the molecular geometry from file formats.

```
mol = IOSystem.from_file('h2o.xyz')
```

1.3.2 Basis Set

MoHa supports basis sets consisting of generally contracted Cartesian Gaussian functions. MoHa is using the same basis set format as NWChem, and the basis sets can be downloaded from the EMSL webpage (https://bse.pnl.gov/bse/portal).

The basis object is a list of list with the follwing structure

$$[IdxxyzCONTRCONTRAtomidx]$$

Idx is the AO index. Atomidx, is the index of the associated atom. #CONTR contains the number of primitive functions in the contracted.

CONTR contains all the information about the primitive functions and have the form:

$$[Nclmn]$$

Inside the integral call, the basisset file is reconstructed into three different arrays, containing the basisset information. The first one is basisidx that have the following form:

$$[primitives \quad loopstartidx]$$

It thus contains the number of primitives in each basisfunction, and what start index it have for loop inside the integral code.

The second array is basisint, that have the following forms:

$$[lmn][lmnatomidx]$$

The first one is for regular integrals and the second one is for derivatives. Both contains all the angular momentum quantum numbers, and the derivative also contains the atom index (used in derivative of VNe).

The last array is basisfloat and have the following forms:

$$[Ncxyz][NcxyzN_{x,+}N_x,N_{y,+}N_y,N_{z,+}N_z,]$$

basisfloat contains the normalization constants, Gaussian exponent and prefacor and the coordinates of the atoms. The second one is again for the derivatives, it contains normalization constants of the differentiated primitives.

1.3.3 Hamilonian

1.3.4 Molecular Integrals

Contains the information about how the integrals are calculated. In the equations in this section the following definitions is used.

$$p = a + b = aba + bPx = aAx + bBxpXAB = AxBx$$

Here a and b are Gaussian exponent factors. Ax and Bx are the position of the Gaussians in one dimension. Further the basisset functions is of Gaussian kind and given as:

$$A(r) = N(xAx)l(yAy)m(zAz)nexp((rA)2)$$

with a normalization constant given as:

$$N = (2)3/4[(8)l + m + nl!m!n!(2l)!(2m)!(2n)!]$$

Boys Function

The Boys function is given as: .. math:

```
Fn(x)=10exp(xt2)t2ndt
```

FUNCTION:

MIcython.boys(m,T) return value Input:

m, subscript of the Boys function T, argument of the Boys function Output:

value, value corrosponding to given m and T

Expansion coefficients

The expansion coefficient is found by the following recurrence relation:

Ei,jt=0,t<0ort>i+j Ei+1,jt=12pEi,jt1+XPAEi,jt+(t+1)Ei,jt+1 Ei,j+1t=12pEi,jt1+XPBEi,jt+(t+1)Ei,jt+1 With the boundary condition that:

E0,00=exp(pX2AB) FUNCTION:

MolecularIntegrals.E(i,j,t,Qx,a,b,XPA,XPB,XAB) return val Input:

i, input values j, input values t, input values Qx, input values a, input values b, input values XPA, input values XPB, input values XAB, input values Output:

val, value corrosponding to the given input

Overlap

The overlap integrals are solved by the following recurrence relation:

$$Si+1,j = XPASij + 12p(iSi1,j + jSi,j1)Si,j+1 = XPBSij + 12p(iSi1,j + jSi,j1)$$

With the boundary condition that:

$$S00 = pexp(X2AB)$$

FUNCTION:

MolecularIntegrals.Overlap(a, b, la, lb, Ax, Bx) return Sij Input:

a, Gaussian exponent factor b, Gaussian exponent factor la, angular momentum quantum number lb, angular momentum quantum number Ax, position along one axis Bx, position along one axis Output:

Sij, non-normalized overlap element in one dimension

Kinetic energy

The kinetic energy integrals are solved by the following recurrence relation:

$$Ti+1,j = XPATi,j + 12p(iTi1,j + jTi,j1) + bp(2aSi+1,jiSi1,j)Ti,j+1 = XPBTi,j + 12p(iTi1,j + jTi,j1) + ap(2bS$$

With the boundary condition that:

$$T00 = [a2a2(X2PA + 12p)]S00$$

FUNCTION:

Kin(a, b, Ax, Ay, Az, Bx, By, Bz, la, lb, ma, mb, na, nb, N1, N2, c1, c2) return Tij, Sij Input:

a, Gaussian exponent factor b, Gaussian exponent factor Ax, position along the x-axis Bx, position along the x-axis Ay, position along the y-axis By, position along the y-axis Az, position along the z-axis Bz, position along the z-axis la, angular momentum quantum number lb, angular momentum quantum number ma, angular momentum quantum number mb, angular momentum quantum number na, angular momentum quantum number nb, angular momentum quantum number N1, normalization constant N2, normalization constant c1, Gaussian prefactor c2, Gaussian prefactor Output:

Tij, normalized kinetic energy matrix element Sij, normalized overlap matrix element

Electron-nuclear attraction

The electron-nuclear interaction integral is given as:

V000ijklmn=2πpti+jEijtuk+lEkluvm+nEmnvRtuv FUNCTION:

MolecularIntegrals.elnuc(P, p, l1, l2, m1, m2, n1, n2, N1, N2, c1, c2, Zc, Ex, Ey, Ez, R1) return Vij Input:

P, Gaussian product p, exponent from Guassian product l1, angular momentum quantum number l2, angular momentum quantum number m1, angular momentum quantum number n1, angular momentum quantum number n2, angular momentum quantum number N1, normalization constant N2, normalization constant c1, Gaussian prefactor c2, Gaussian prefactor Zc, Nuclear charge Ex, expansion coefficients Ey, expansion coefficients Ez, expansion coefficients R1, hermite coulomb integrals Output:

Vij, normalized electron-nuclei attraction matrix element

Electron-electron repulsion

The electron-electron repulsion integral is calculated as:

gabcd=tl1+l2Eabtum1+m2Eabuvn1+n2Eabvτl3+l4Ecd$\tau\nu$m3+m4Ecdνn3+n4Ecd(1)τ+ν+2π5/2pqp+qRt+τ,u+ν,v+(α,RPQ) FUNCTION:

MIcython.elelrep(p, q, l1, l2, l3, l4, m1, m2, m3, m4, n1, n2, n3, n4, N1, N2, N3, N4, c1, c2, c3, c4, E1, E2, E3, E4, E5, E6, Rpre) return Veeijkl Input:

p, Gaussian exponent factor from Gaussian product q, Gaussian exponent factor from Gaussian product l1, angular momentum quantum number l2, angular momentum quantum number l3, angular momentum quantum number l4, angular momentum quantum number m1, angular momentum quantum number m2, angular momentum quantum number m3, angular momentum quantum number m4, angular momentum quantum number n1, angular momentum quantum number n2, angular momentum quantum number n3, angular momentum quantum number n4, angular momentum quantum number N1, normalization constant N2, normalization constant N3, normalization constant N4, normalization constant c1, Gaussian prefactor c2, Gaussian prefactor c3, Gaussian prefactor c4, Gaussian prefactor E1, expansion coefficient E2, expansion coefficient E3, expansion coefficient E4, expansion coefficient E5, expansion coefficient E6, expansion coefficient Rpre, hermite coulomb integral Output:

Veeijkl, normalized electron-electron repulsion matrix element

1.3.5 Build Hamilton

To build a Hamiltonian object, MoHa can load the molecular geometry and and basis from file format.

```
mol,orbs = IOSystem.from_file('h2o.xyz','sto-3g.nwchem')
ham = Hamiltonian.build(mol,orbs)
```

1.4 Model System

In terms of second quantisation operators, a general Hamiltonian can be written as

$$H = -\sum_{ij} t_{ij} \hat{c}_i^\dagger \hat{c}_j + \frac{1}{2} \sum_{ijkl} V_{ijkl} \hat{c}_i^\dagger \hat{c}_k^\dagger \hat{c}_l \hat{c}_j$$

where $c_{i,\sigma}$ is the destruction operator for an electron at site i and spin σ, $c_{i,\sigma}^\dagger$ is the creation operator for an electron at site i and spin σ.

Model Hamiltonians geared to simulate the key physics of notoriously complicated complete Hamiltonians of large-scale interacting systems.

1.4.1 Lattice

Lattice is an periodic array of discrete points in space generated by a set of discrete translation operations described by:

$$\vec{R} = n_1\vec{a}_1 + n_2\vec{a}_2 + n_3\vec{a}_3$$

where n_i are any integers and $\vec{a}_i$ are known as the primitive vectors which lie in different directions and span the lattice. The discrete points can be atoms, ions, electronic site, spin one-half site, spinless fermion site, etc.

Generaly speaking, to build a The `Lattice` object, we need first define a `Cell` object. `Cell` has attribute of dimension and three primitive vectors $\vec{a}_1$ $\vec{a}_2$ and $\vec{a}_3$. To initialize a cell:

```
cell = Cell(1,[d,0.,0.],[0.,0.,0.],[0.,0.,0.])
```

Then we can add arbitrary sites to this cell by assign coordinate and label of the site:

```
cell.add_site(LatticeSite([0.,0.,0.],'A'))
```

Then we add bonds to this cell by assign the integer coordinate of first cell, index of first site in first cell and the integer coordinate of second cell, index of the second site in second cell:

```
cell.add_bond(LatticeBond([0,0,0],0,[1,0,0],0))
```

Finally complete it by assign the cell and shpe to the lattice to `Lattice`:

```
lattice = Lattice(cell,[4,1,1])
```

It may not be immediately obvious what this code does. Fortunately, `Lattice` objects have a convenient `Lattice.plot()` method to easily visualize the constructed lattice.

Linear lattice

Starting from the basics, we'll build a simple linear lattice with only one site.

Listing 1: /data/examples/modelsystem/linear.py

```
from moha import *

d = 1.0 # unit cell length

def linear(d):
    #define a 1D linear lattice cell with vectors a1 a2 and a3
    cell = Cell(1,[d,0.,0.],[0.,0.,0.],[0.,0.,0.])
    #add a site labeled 'A' at positon [0.,0.,0.]
    cell.add_site(LatticeSite([0.,0.,0.],'A'))
    #add a bond from first site in cell [0,0,0] to first site in cell [1,0,0]
    cell.add_bond(LatticeBond([0,0,0],0,[1,0,0],0))
    #buld a lattice of shape [4,1,1] with cell we defined
    lattice = Lattice(cell,[4,1,1])
    return lattice

lattice = linear(d)
#plot the lattice we constructed
lattice.plot()
```

Visualize the lattice by `Lattice.plot()` method.

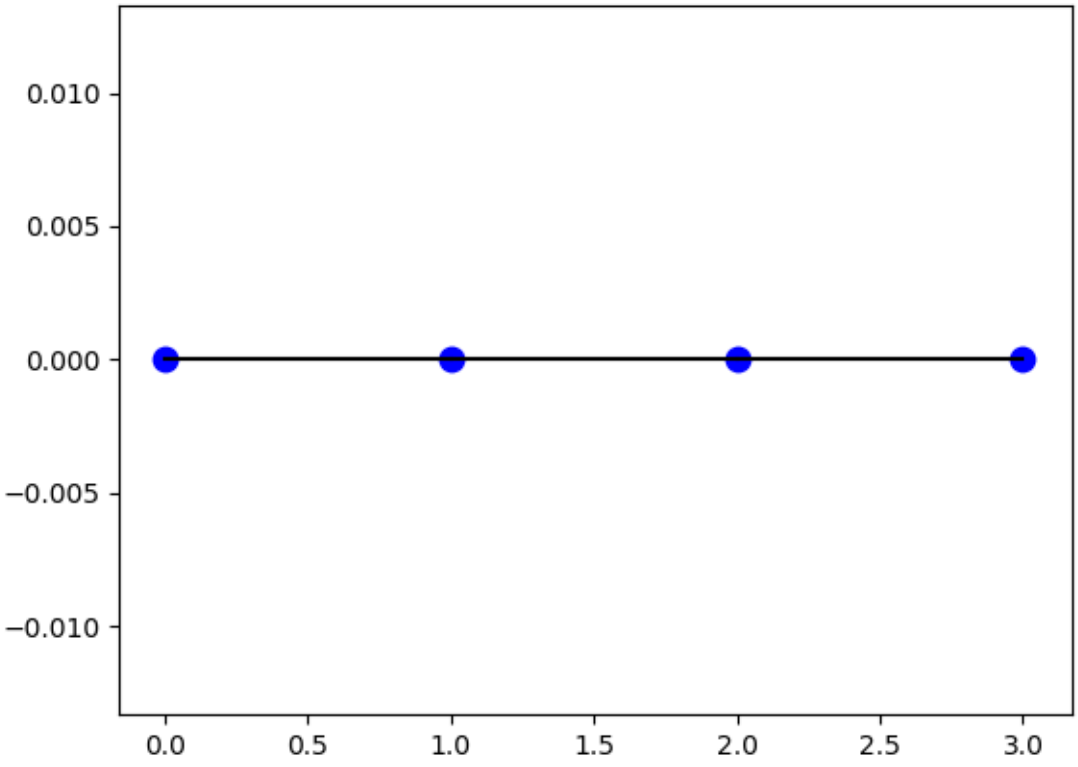
Two sites linear lattice

The next example is also linear lattice but a slightly more complicated, the cell of the lattice is consist of two different sites.

Listing 2: /data/examples/modelsystem/two_sites_linear.py

```
from moha import *

d = 1.0      # unit cell length
```

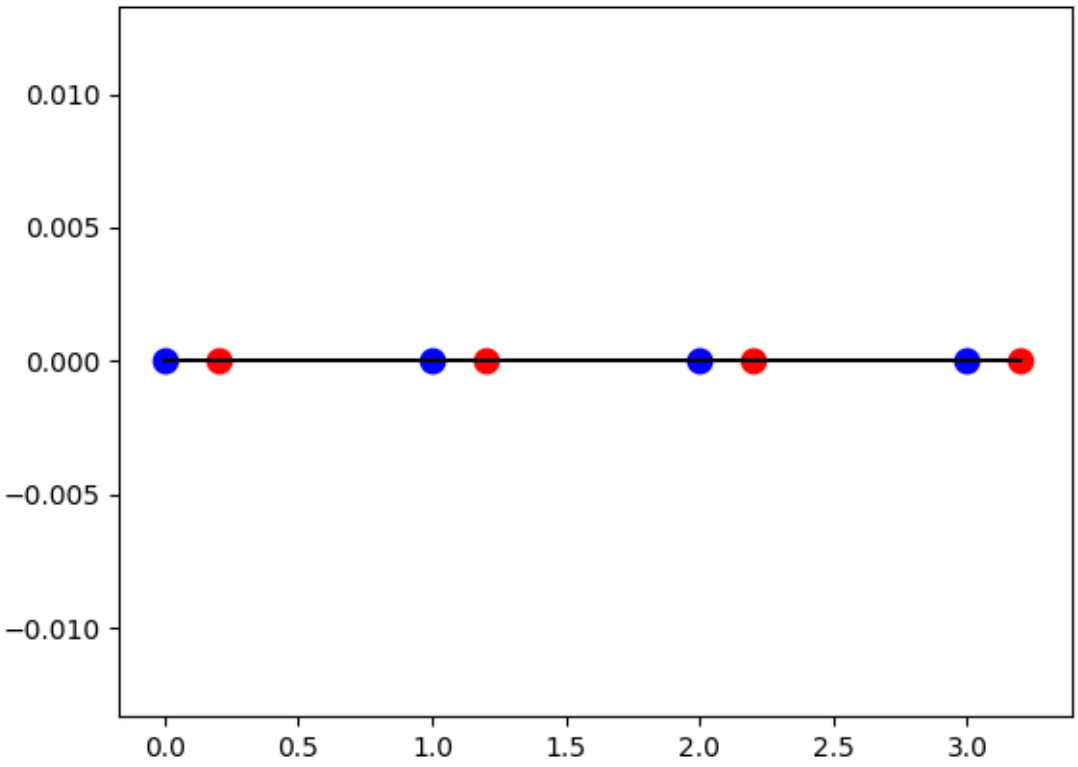
(continues on next page)


0.010
0.005
0.000
−0.005
−0.010
0.0
0.5
1.0
1.5
2.0
2.5
3.0

(continued from previous page)

```
d_ab = 0.2  # distance between site A and B

def linear(d,d_ab):
    #define a 1D linear lattice cell with vectors a1 a2 and a3
    cell = Cell(1,[d, 0., 0.],[0., 0., 0.],[0., 0., 0.])
    #add a site labeled 'A' at positon [0., 0., 0.]
    cell.add_site(LatticeSite([0., 0., 0.], 'A'))
    #add a site labeled 'B' at positon [d_ab, 0., 0.]
    cell.add_site(LatticeSite([d_ab, 0., 0.],'B'))
    #add a bond from first site in cell [0,0,0] to second site in cell [0,0,0]
    cell.add_bond(LatticeBond([0,0,0],0,[0,0,0],1))
    #add a bond from second site in cell [0,0,0] to first site in cell [1,0,0]
    cell.add_bond(LatticeBond([0,0,0],1,[1,0,0],0))
    #buld a lattice of shape [4,1,1] with cell we defined
    lattice = Lattice(cell,[4,1,1])
    return lattice
```

Visualize the lattice by `Lattice.plot()` method.



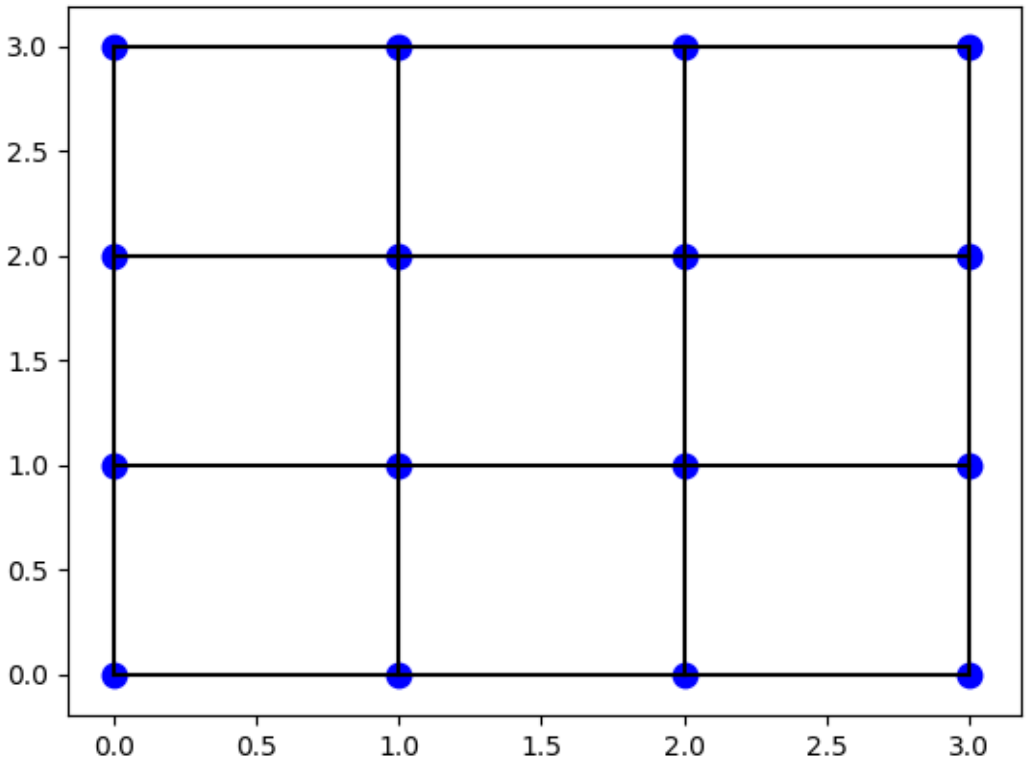
Square lattice

From 1D to 2D, the most basic example square lattice.

Listing 3: /data/examples/modelsystem/square.py

```
from moha import *

a = 1.0 #unit cell length

def square(a):
    #define a 2D square lattice cell with vectors a1 a2 and a3
    cell = Cell(2,[a, 0., 0.],[0., a, 0.],[0., 0., 0.])
    #add a site labeled 'A' at positon [0.,0.,0.]
    cell.add_site(LatticeSite([0.,0.,0.],'A'))
    #add a bond from first site in cell [0,0,0] to first site in cell [1,0,0]
    cell.add_bond(LatticeBond([0,0,0],0,[1,0,0],0))
    #add a bond from first site in cell [0,0,0] to first site in cell [0,1,0]
    cell.add_bond(LatticeBond([0,0,0],0,[0,1,0],0))
    #buld a lattice of shape [4,4,1] with cell we defined
    lattice = Lattice(cell,[4,4,1])
    return lattice

lattice = square(a)
#plot the lattice we constructed
lattice.plot()
```

Visualize the lattice by `Lattice.plot()` method.



Graphene lattice

Graphene lattice is a more general example:

- It is two dimension.
- The cell of graphene contain two sites.
- The primitive vectors of the cell are non-orthogonal.

Listing 4: /data/examples/modelsystem/graphene.py

```
from moha import *
from math import sqrt

a = 0.24595     # unit cell length
a_cc = 0.142    # carbon carbon distance

def graphene(a,a_cc):


    #define a 2D graphene lattice cell with vectors a1 a2 and a3
    cell = Cell(2,[[a,0.,0.],[a/2,a/2*sqrt(3),0.],[0.,0.,0.]])


    #add a site labeled 'A' at positon [0., -a_cc/2., 0.]
    cell.add_site(LatticeSite([0., -a_cc/2., 0.],'A'))
    #add a site labeled 'B' at positon [0., a_cc/2., 0.]
    cell.add_site(LatticeSite([0., a_cc/2., 0.],'B'))

    #add a bond from first site in cell [0,0,0] to second site in cell [0,0,0]
    cell.add_bond(LatticeBond([0,0,0],0,[0,0,0],1))
    #add a bond from second site in cell [0,0,0] to first site in cell [0,1,0]
    cell.add_bond(LatticeBond([0,0,0],1,[0,1,0],0))
    #add a bond from first site in cell [0,0,0] to second site in cell [1,-1,0]
    cell.add_bond(LatticeBond([0,0,0],0,[1,-1,0],1))

    #buld a lattice of shape [4,4,1] with cell we defined
    lattice = Lattice(cell,[4,4,1])
    return lattice

lattice = graphene(a,a_cc)
#plot the lattice we constructed
lattice.plot()
```

Visualize the lattice by `Lattice.plot()` method.

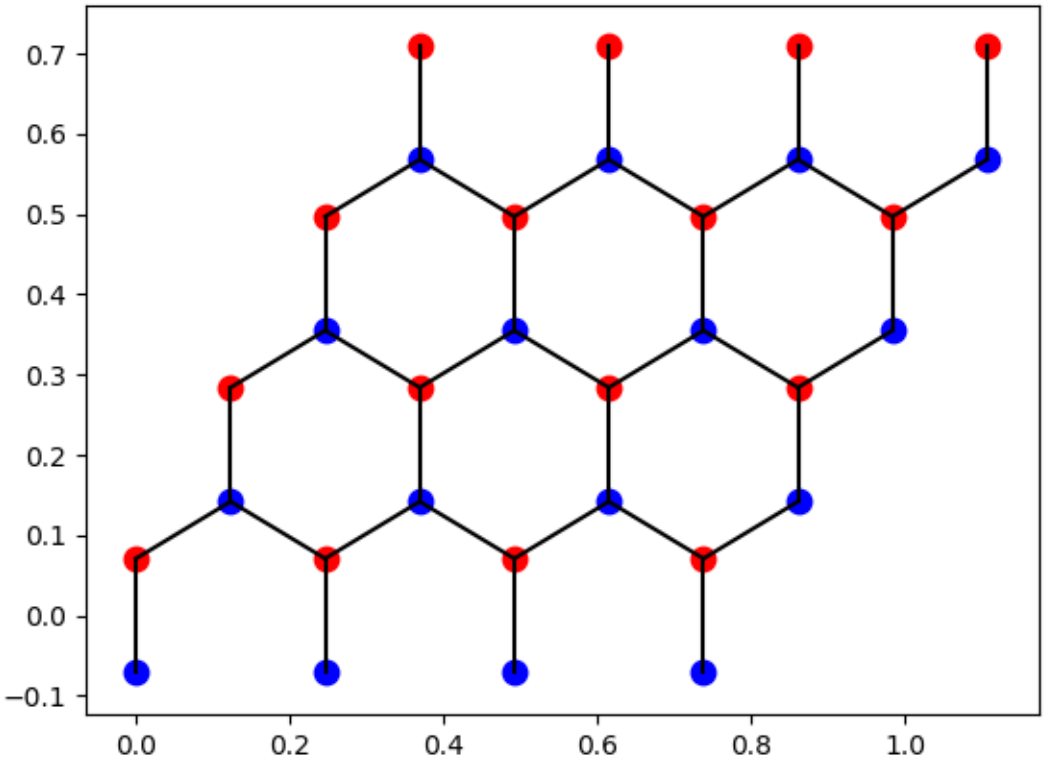
Cubic lattice

Final example is a 3d cubic lattice.

Listing 5: /data/examples/modelsystem/cubic.py

```
from moha import *

d = 1.0     # unit cell length

def cube(d):
```

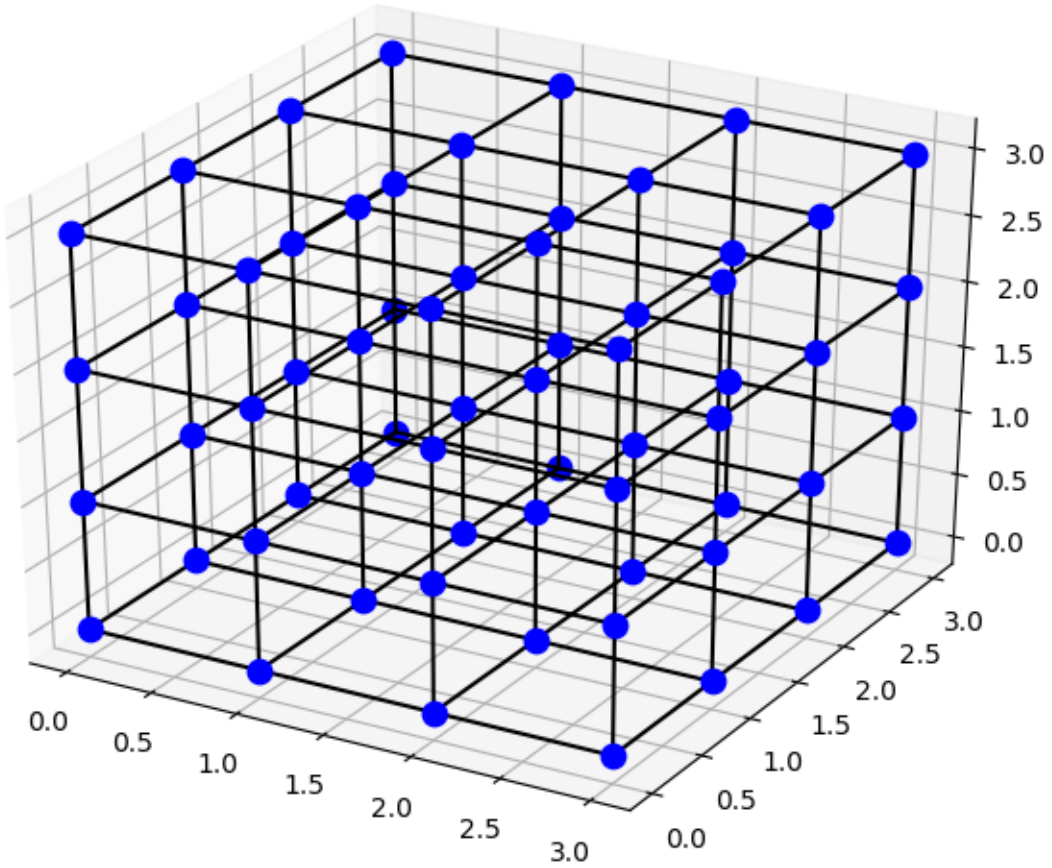
(continues on next page)


0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
−0.1
0.0
0.2
0.4
0.6
0.8
1.0

(continued from previous page)

```
    #define a 3D cubic lattice cell with vectors a1 a2 and a3
    cell = Cell(3,[d, 0., 0.],[0., d, 0.],[0., 0., d])

    #add a site labeled 'A' at positon [0.,0.,0.]
    cell.add_site(LatticeSite([0.,0.,0.],'A'))

    #add a bond from first site in cell [0,0,0] to second site in cell [0,0,0]
    cell.add_bond(LatticeBond([0,0,0],0,[1,0,0],0))
    #add a bond from second site in cell [0,0,0] to first site in cell [0,1,0]
    cell.add_bond(LatticeBond([0,0,0],0,[0,1,0],0))
    #add a bond from second site in cell [0,0,0] to first site in cell [0,1,0]
    cell.add_bond(LatticeBond([0,0,0],0,[0,0,1],0))

    #buld a lattice of shape [4,4,4] with cell we defined
    lattice = Lattice(cell,[4,4,4])
    return lattice

lattice = cube(d)
#plot the lattice we constructed
lattice.plot()
```

Visualize the lattice by `Lattice.plot()` method.



1.4.2 Model Hamiltonian

Analogy to the construction of molecular Hamiltonian, model hamiltonian usually set up in three steps.

- First, construct a lattice.
- Second, generate a basis set for the lattice.To simplifies the mathematics, In what follows we will assume that the basis set is orthonormal.
- Finally, compute all kinds of one body terms and two body terms with that basis to define a Hamiltonian.

With the definition of lattice and site object done, to construct a model Hamiltonian, we need:

- Initialize the model Hamiltonian by assign lattice and site

```
moha = ModelHamiltonian(lattice,site)
```

- Then add one and two body interaction terms

```
moha.add_operator(OneBodyTerm(['c_dag','c'],[0,0,0],0,[0,0,0],0,-1.0))

moha.add_operator(TwoBodyTerm(['n','n'],[0,0,0],0,[0,0,0],0,2.0))
```

You can check the one body interaction matrix and two body interaction tensor by

```
moha.one_body_matrix

moha.two_body_tensor
```

Fermion Model

- Tight Binding Model

In solid-state physics, the tight-binding model (or TB model) is an approach to the calculation of electronic band structure using an approximate set of wave functions based upon superposition of wave functions for isolated atoms located at each atomic site.

The Hamiltonian is:

$$\hat{H} = -t \sum_{<i,j>\sigma} \hat{c}^{\dagger}_{i,\sigma} \hat{c}_{j,\sigma} - \mu \sum_{i\sigma} \hat{c}^{\dagger}_{i\sigma} \hat{c}_{i\sigma}$$

Often one considers models with only nearest neighbour terms, $< ij >$ indicates nearest neighbours. And one takes $t_{ij} = t$ if i and j are at nearest neighbour sites, and $t_{ij} = 0$ otherwise, μ is the chemical potential.

To represent the Hamilontion in the matrix form, we ignore the spin of the Hamiltonian and transform the Hamiltonion to

$$\hat{H} = -t \sum_{<i,j>} \hat{c}^{\dagger}_{i} \hat{c}_{j} - \mu \sum_{i} \hat{c}^{\dagger}_{i} \hat{c}_{i}$$

Listing 6: /data/examples/modelsystem/tight_binding.py

```
from moha import *
from math import sqrt

a = 0.24595      # unit cell length
```

(continues on next page)

(continued from previous page)

```
a_cc = 0.142    # carbon carbon distance

def graphene(a,a_cc):


    #define a 2D graphene lattice cell with vectors a1 a2 and a3
    cell = Cell(2,[[a,0.,0.],[a/2,a/2*sqrt(3),0.],[0.,0.,0.]])


    #add a site labeled 'A' at positon [0.,-a_cc/2.,0.]
    cell.add_site(LatticeSite([0., -a_cc/2., 0.],'A'))
    #add a site labeled 'B' at positon [0.,-a_cc/2.,0.]
    cell.add_site(LatticeSite([0., a_cc/2., 0.],'B'))

    #add a bond from first site in cell [0,0,0] to second site in cell [0,0,0]
    cell.add_bond(LatticeBond([0,0,0],0,[0,0,0],1))
    #add a bond from second site in cell [0,0,0] to first site in cell [0,1,0]
    cell.add_bond(LatticeBond([0,0,0],1,[0,1,0],0))
    #add a bond from first site in cell [0,0,0] to second site in cell [1,-1,0]
    cell.add_bond(LatticeBond([0,0,0],0,[1,-1,0],1))

    #buld a lattice of shape [4,4,1] with cell we defined
    lattice = Lattice(cell,[4,4,1])
    return lattice

lattice = graphene(a,a_cc)
#use spinless fermion site as basis
site = SpinlessFermionSite()
#build model hamiltonian
mh = ModelHamiltonian(lattice,site)
#add on-site term of site A
mh.add_operator(OneBodyTerm(['c_dag','c'],[0,0,0],0,[0,0,0],0,-1.0))
#add on-site term of site B
mh.add_operator(OneBodyTerm(['c_dag','c'],[0,0,0],1,[0,0,0],1,-2.0))
#add hopping term of site between site A and B
mh.add_operator(OneBodyTerm(['c_dag','c'],[0,0,0],0,[0,0,0],1,-4.0))
mh.add_operator(OneBodyTerm(['c_dag','c'],[0,0,0],1,[0,1,0],0,-4.0))
mh.add_operator(OneBodyTerm(['c_dag','c'],[0,0,0],0,[1,-1,0],1,-4.0))
```

- Hubbard Model

The Hubbard model is an approximate model used, especially in solid-state physics, to describle the transition between conducting and insulating systems. The basic hubbard model have only two Hamiltonian is the simplest model of interacting particles on a lattice and reads.

The Hamiltonian is:

$$\hat{H} = -t \sum_{<i,j>\sigma} \hat{c}^{\dagger}_{i,\sigma} \hat{c}_{j,\sigma} + U \sum_{i} \hat{n}_{i,\uparrow} \hat{n}_{i,\downarrow}$$

where the first term is a one-electron term and accounts for the nearest-neighbor hopping, while the second term is the repulsive on-site interaction. The t and U are user specified parameters and σ is the electron spin.

To represent the Hamilontion in the matrix form, we ignore the spin of the Hamiltonian and transform the Hamiltonion to

$$\hat{H} = -t \sum_{<i,j>} \hat{c}^{\dagger}_{i} \hat{c}_{j} + U \sum_{i} \hat{n}_{i} \hat{n}_{i}$$

Listing 7: /data/examples/modelsystem/hubbard.py

```
from moha import *
from math import sqrt

a = 1.0 #unit cell length

def square(a):
    #define a 2D square lattice cell with vectors a1 a2 and a3
    cell = Cell(2,[a, 0., 0.],[0., a, 0.],[0., 0., 0.])
    #add a site labeled 'A' at positon [0.,0.,0.]
    cell.add_site(LatticeSite([0.,0.,0.],'A'))
    #add a bond from first site in cell [0,0,0] to first site in cell [1,0,0]
    cell.add_bond(LatticeBond([0,0,0],0,[1,0,0],0))
    #add a bond from first site in cell [0,0,0] to first site in cell [0,1,0]
    cell.add_bond(LatticeBond([0,0,0],0,[0,1,0],0))
    #buld a lattice of shape [4,4,1] with cell we defined
    lattice = Lattice(cell,[4,4,1])
    return lattice

lattice = square(a)
#use spinless fermion site as basis
site = SpinlessFermionSite()
#build model hamiltonian
mh = ModelHamiltonian(lattice,site)
#add hopping term of site between nearest neighbour A and B
mh.add_operator(OneBodyTerm(['c_dag','c'],[0,0,0],0,[1,0,0],0,-1.0))
#add two body term of site between site A and B
mh.add_operator(TwoBodyTerm(['n','n'],[0,0,0],0,[0,0,0],0,2.0))
```

Spin Model

Quantum spins are a complex object to deal with in many-body physics, it's niether a canonical fermions or bosons. However, we can perform the Jordan–Wigner transformation that maps spin operators onto fermionic creation and annihilation operators

where

$$S_j^z = c_j^\dagger c_j - \frac{1}{2}$$

$$S_j^+ = c_j^\dagger e^{i\pi \sum_{l<j} n_l}$$

$$S_j^- = c_j e^{-i\pi \sum_{l<j} n_l}$$

- Ising Model

The Ising model is a mathematical model of ferromagnetism in statistical mechanics. The model consists of discrete variables that represent magnetic dipole moments of atomic spins that can be in one of two states (+1 or 1), allowing each spin to interact with its neighbors.

The Hamiltonian is:

$$H = \sum_{<i,j>\sigma} -JS_i^z S_j^z$$

After Jordan–Wigner transformation, it becomes:

$$H = J\sum_i n_i - J\sum_i n_{i+1} n_i$$

Listing 8: /data/examples/modelsystem/ising.py

```
from moha import *

d = 1.0 # unit cell length
J = 1.0 # spin interaction parameter

def linear(d):
    #define a 1D linear lattice cell with vectors a1 a2 and a3
    cell = Cell(1,[d,0.,0.],[0.,0.,0.],[0.,0.,0.])
    #add a site labeled 'A' at positon [0.,0.,0.]
    cell.add_site(LatticeSite([0.,0.,0.],'A'))
    #add a bond from first site in cell [0,0,0] to first site in cell [1,0,0]
    cell.add_bond(LatticeBond([0,0,0],0,[1,0,0],0))
    #buld a lattice of shape [4,1,1] with cell we defined
    lattice = Lattice(cell,[4,1,1])
    return lattice

lattice = linear(d)
site = SpinlessFermionSite()
moha = ModelHamiltonian(lattice,site)
moha.add_operator(OneBodyTerm(['n'],[0,0,0],0,[0,0,0],0,J))
moha.add_operator(TwoBodyTerm(['n','n'],[0,0,0],0,[1,0,0],0,-J))
```

- Heisenberg Model

The Heisenberg model is a statistical mechanical model used in the study of critical points and phase transitions of magnetic systems, in which the spins of the magnetic systems are treated quantum mechanically.

The Hamiltonian is:

$$H = J \sum_{<i,j>} \vec{S}_i \vec{S}_j = J \sum_{<ij>} \left[S_i^z S_j^z + \frac{1}{2} \left(S_i^+ S_j^- + S_i^- S_j^+ \right) \right]$$

where

$$\hat{S}_i^+ = \hat{S}_i^x + i\hat{S}_i^y$$

$$\hat{S}_i^- = \hat{S}_i^x - i\hat{S}_i^y$$

$$\hat{S}_i^z = \frac{1}{2}(\hat{S}_i^+ \hat{S}_i^- - \hat{S}_i^- \hat{S}_i^+)$$

To perform the Jordan-Wigner transformation

$$H = -\frac{J}{2} \sum_i \left(c_{i+1}^\dagger c_i + c_i^\dagger c_{i+1} \right) + J \sum_i n_i - J \sum_i n_{i+1} n_i$$

Listing 9: /data/examples/modelsystem/heisenberg.py

```
from moha import *

d = 1.0 # unit cell length
J = 1.0 # spin interaction parameter

def linear(d):
```

(continues on next page)

(continued from previous page)

```
    #define a 1D linear lattice cell with vectors a1 a2 and a3
    cell = Cell(1,[d,0.,0.],[0.,0.,0.],[0.,0.,0.])
    #add a site labeled 'A' at positon [0.,0.,0.]
    cell.add_site(LatticeSite([0.,0.,0.],'A'))
    #add a bond from first site in cell [0,0,0] to first site in cell [1,0,0]
    cell.add_bond(LatticeBond([0,0,0],0,[1,0,0],0))
    #buld a lattice of shape [4,1,1] with cell we defined
    lattice = Lattice(cell,[4,1,1])
    return lattice

lattice = linear(d)
site = SpinlessFermionSite()
moha = ModelHamiltonian(lattice,site)
moha.add_operator(OneBodyTerm(['c_dag','c'],[0,0,0],0,[1,0,0],0,-J/2.))
moha.add_operator(OneBodyTerm(['n'],[0,0,0],0,[0,0,0],0,J))
moha.add_operator(TwoBodyTerm(['n','n'],[0,0,0],0,[1,0,0],0,-J))
```

- XXZ Model

The Hamiltonian is:

$$H = \sum_{<ij>} J_z S_i^z S_j^z + \frac{J}{2}\left(S_i^+ S_j^- + S_i^- S_j^+\right)$$

To perform the Jordan-Wigner transformation

$$H = -\frac{J}{2}\sum_i \left(c_{i+1}^\dagger c_i + c_i^\dagger c_{i+1}\right) + J_z \sum_i n_i - J_z \sum_i n_{i+1} n_i$$

- XY Model

The Hamiltonian is:

$$H = \sum_{<ij>} \frac{J}{2}\left(S_i^+ S_j^- + S_i^- S_j^+\right)$$

To perform the Jordan-Wigner transformation

$$H = -\frac{J}{2}\sum_i \left(c_{i+1}^\dagger c_i + c_i^\dagger c_{i+1}\right)$$

1.4.3 Program reference

Non-relativistic Hartree-Fock

1.5 SCF

To begin a calculation with MoHa, the first step is to build a Hamiltonian of a system, either molecular system or model system. In most cases, we need to build a molecular system

In terms of second quantisation operators, a general Hamiltonian can be written as

$$H = -\sum_{ij} t_{ij} \hat{c}_i^\dagger \hat{c}_j + \frac{1}{2}\sum_{ijkl} V_{ijkl} \hat{c}_i^\dagger \hat{c}_k^\dagger \hat{c}_l \hat{c}_j$$

The construction of molecular Hamiltonian usually set up in three steps.

- First, construct a molecular geometry.
- Second, generate a Gaussian basis set for the molecular.
- Finally, compute all kinds of one body terms and two body terms with that basis to define a Hamiltonian.

1.6 Properties

This section containts information about atomic and molecular properties that can be calculated.

1.6.1 Population Analysis/Atomic Charges

1.6.2 Multipole Moment

1.6.3 Random-Phase Approximation Excitation energy

1.7 API Doucmentation

This part of the documentation is generated from the docstrings in the source code.

1.8 API Doucmentation

To begin a calculation with MoHa, the first step is to build a Hamiltonian of a system, either molecular system or model system. In most cases, we need to build a molecular system

In terms of second quantisation operators, a general Hamiltonian can be written as 1 .. math:

```
H = - \sum_{ij} t_{ij}\hat{c}^{\dagger}_{i}\hat{c}_{j} + \frac{1}{2} \sum_{ijkl}
V_{ijkl}\hat{c}^{\dagger}_{i}\hat{c}^{\dagger}_{k}\hat{c}_{l}\hat{c}_{j}
```

The construction of molecular Hamiltonian usually set up in three steps.

- First, construct a molecular geometry.
- Second, generate a Gaussian basis set for the molecular.
- Finally, compute all kinds of one body terms and two body terms with that basis to define a Hamiltonian.

1.8.1 Molecule

fdfdsfs

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex

PYTHON MODULE INDEX

m