

Graph based Block Digonalization 코드 주석 정리

최윤희

1 분자 (화합물) 정의

```
1 geometry = [('H', (0., 0., 0.)),
2             ('H', (dist, 0., 0.)),
3             ('H', (2*dist, 0., 0.)),
4             ('H', (3*dist, 0., 0.))] # Angstrom
5 basis = 'sto-3g'
6 mol = MolecularData(geometry, basis, multiplicity=1, charge=0)
7 # qiskit 의 molecularinfo 와 같이 분자의 정보를 담고있는 클래스를 생성 이때 필요한 정보는
  → 기하학적인 구조 (geometry), basis, 시스템의 multiplicity 와 총 전하량 등의 정보가 필
  → 요하다.
```

* geometry : 원자의 종류와 기하학적인 구조를 튜플로써 저장 (' 원소종류', (x,y,z))

* basis : 전자의 atomic orbital 을 표현할 basis 설정 (sto3g, 6-31g,...)

* MolecularData: qiskit 의 molecularinfo 와 같이 분자의 정보를 담고있는 클래스

이는 객체는 분자 (화합물) 의 정보를 담고있는 객체로써, 이후 이 객체로부터 전자의 개수나, scf 계산을 수행할것이다. 이번 논문에서 사용한 화합물은 H2 Cluster 로써 아래와같이 모든 수소원자가 같은 거리 만큼 떨어져있는 구조를 의미한다.

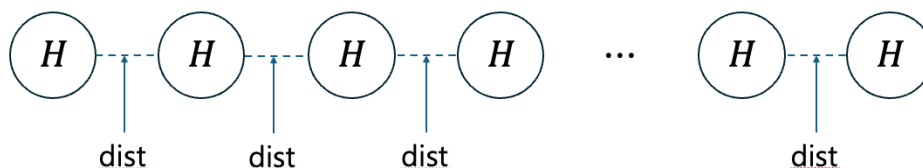


Figure 1: H2 Cluster

2 FCI 행렬 생성

```
1 mol = run_pyscf(mol_input, run_scf=1, run_fci=0)
2 # mol_input : 앞서 MolecularData 클래스를 이용해 생성된 객체 분자의 여러 정보를 담고있다.
3 # run_pyscf : mol_input 을 바탕으로 계산을 수행 2nd Quantized 헤밀토니안을 얻기위한 HF 계
  ↳ 산을 수행한다.
4 # 이때 옵션으로 여기서 FCI 계산을 수행할수도 있으나, 이는 Cost 가 매우커서 수행하지 않는다.
5 ham_int = mol.get_molecular_hamiltonian()
6 # pyscf 계산을 수행한 이후 정보를 담고있는 mol 이라는 객체로 부터 1 차/2 차 적분에 대한 정
  ↳ 보를 가져온다.
7 # (*) 이후 객체 정보에 대한 추가설명
8 ham_fci = get_fermion_operator(ham_int)
9 # 1 차/2 차 적분에 대한 정보를 담고있는 ham_int 로 부터 2nd Quantized 된 헤밀토니안을 가져
  ↳ 온다.
10 # (**) 이후 객체 정보에 대한 추가설명
11
12 H = get_number_preserving_sparse_operator(
13 # 전자수 와 스핀량 보존을 포함하는 FCI 행렬을 만들기 위한 함수
14 fermion_op=ham_fci, # 사용할 2nd Quantized 헤밀토니안
15 num_qubits=mol.n_qubits, # 총 스핀오비탈 개수 openFermion 패키지에서는 이를 qubit 라는
  ↳ 이름으로 사용하지만, 같은 의미.
16 num_electrons=mol.n_electrons, # 시스템의 전자수
17 spin_preserving=True) # 시스템의 multiplicity 가 보존되는 Determinant 만을 사용
18 H_real = H.real # 행렬의 각 원소는 두 Determinant 로 기술되는 내적이며 이는 에너지에 대응
  ↳ 되는 값이므로 실수값을 갖게되므로, 이후 그래프 연산을 위해 실수로 변환
19 # (***) 이후 객체 정보에 대한 추가설명
```

2.1 (*) mol.get_molecular_hamiltonian()

1 차, 2 차 적분의 결과를 아래와같이 저장 한다.

```
( ) 0.7137539936876182
((0, 1), (0, 0)) -1.2524635735648981
((1, 1), (1, 0)) -1.2524635735648981
((2, 1), (2, 0)) -0.4759487152209644
((3, 1), (3, 0)) -0.4759487152209644
((0, 1), (0, 1), (0, 0), (0, 0)) 0.33724438317841876
((0, 1), (0, 1), (2, 0), (2, 0)) 0.09064440410574796
((0, 1), (1, 1), (1, 0), (0, 0)) 0.33724438317841876
((0, 1), (1, 1), (3, 0), (2, 0)) 0.09064440410574796
((0, 1), (2, 1), (0, 0), (2, 0)) 0.09064440410574796
((0, 1), (2, 1), (2, 0), (0, 0)) 0.3317340482117839
((0, 1), (3, 1), (1, 0), (2, 0)) 0.09064440410574796
((0, 1), (3, 1), (3, 0), (0, 0)) 0.3317340482117839
((1, 1), (0, 1), (0, 0), (1, 0)) 0.33724438317841876
```

Figure 2: molecular Hamiltonian

* () : 가장 위의 공란은 Constant 값 (여기서는 핵간 척력)

* ((0,1),(0,0)) : 튜플의 튜플로 정의 된다. 안의 튜플은 순서대로 (연산이 수행되는 오비탈 idx, 생성 (1) 인지, 소멸 (0) 인지) 결국 second Quantization 에서 1 차여기항에 대응된다.

* ((0, 1), (0, 1), (0, 0), (0, 0)) : 튜플 4 개로 정의 된다. 안의 튜플은 위에서와 같은 정보를 사용. 결국 second Quantization 에서 2 차여기항에 대응된다.

2.2 (**) get_fermion_operator(ham_int)

1 차, 2 차 적분의 결과를 아래와같이 fermionic 생성/소멸 연산자의 표기법을 통해 2nd Quantized 해밀토니안으로 표현한다.

```
0.7137539936876182 [ ] +
-1.2524635735648981 [0^ 0] +
0.33724438317841876 [0^ 0^ 0 0] +
0.09064440410574796 [0^ 0^ 2 2] +
0.33724438317841876 [0^ 1^ 1 0] +
0.09064440410574796 [0^ 1^ 3 2] +
0.09064440410574796 [0^ 2^ 0 2] +
0.3317340482117839 [0^ 2^ 2 0] +
0.09064440410574796 [0^ 3^ 1 2] +
0.3317340482117839 [0^ 3^ 3 0] +
0.33724438317841876 [1^ 0^ 0 1] +
0.09064440410574796 [1^ 0^ 2 3] +
```

Figure 3: fermionic op

여기서,

[] : Constant 값 (여기서는 핵간 척력)

$$[i^{\wedge}j] : a_i^{\dagger}a_j$$

$$[i^{\wedge}j^{\wedge}k\ l] : a_i^{\dagger}a_j^{\dagger}a_k a_l$$

2.3 (***) get_number_preserving_sparse_operator()

아래와같이 희소행렬 표현으로 FCI 행렬을 구성한다.

```
<Compressed Sparse Column sparse matrix of dtype 'float64'
  with 8 stored elements and shape (4, 4)>
  Coords      Values
(0, 0)      -1.1166843870853407
(3, 0)       0.18128880821149593
(1, 1)      -0.3511901986746764
(2, 1)      -0.18128880821149593
(1, 2)      -0.18128880821149593
(2, 2)      -0.3511901986746764
(0, 3)       0.18128880821149593
(3, 3)       0.4592503306687161
```

Figure 4: FCI matrix

3 FCI 행렬 -> 그래프로 변경

```
1 def sparse_to_graph(A):
2     """
3     희소행렬 A -> NetworkX Graph/DiGraph 변환
4
5     Returns
6     -----
7     G : nx.Graph
8     """
9
10    if not issparse(A):
11        raise ValueError("A must be a scipy.sparse matrix")
12
13    # COO 로 변환
14    A = A.tocoo(copy=True)
15    # COO 에서는 값을 아래와같이 3 개의 어레이로 저장
16    # data : [3,4,2,7]
17    # row : [0,0,1,3]
18    # col : [2,4,1,3]
19    # 그리고 같은 인덱스 순서대로 0 행 2 열에는 3 이라는 값이 있고 마찬가지로 총 4 개의 원소
20    ↪ 가 있는 행렬을 표현한다.
21
22    G = nx.Graph()
23    # NetworkX 패키지의 Graph 클래스를 불러온다.
24    G.add_nodes_from(range(A.shape[0])) # 노드: 0..n-1
25    # A(coo matrix) 의 길이, 즉 행렬의 원소의 개수만큼의 node 를 만든다.
26    vals = np.abs(A.data)
27    # 행렬의 데이터 (이경우 hij) 를 하나의 어레이로 만든다.
28
29    # 동일 엣지 중복 합치기 (무향일 때 i<j 묵기)
30    from collections import defaultdict
31    edges = defaultdict(float)
32    # 딕셔너리를 만들때, 특정 키가 주어졌을때, 그 키가 만약 존재하지 않다면 0.0 이라는 값을
33    ↪ 그 키로 하는 value 를 만든다.
34    for i, j, v in zip(A.row, A.col, vals):
35        # 행 (i), 열 (j), 행렬원소 (hij)
36        if i == j :
37            continue
38            # 대각원소는 자기와의 간선이므로, 이는 고려하지 않는다.
39        key = (min(i, j), max(i, j))
40        # key 를 만들어 이를 이용해 간선 (edges) 를 만들것이다.
41        # 그런데, '무향' 그래프 에선 두 노드 사이의 방향이 없으므로
```

```

40     #  $h_{ij}$  와  $h_{ji}$  를 같은 키로 만들어 위와같이 정의한다.
41     edges[key] += 0.5*float(v)
42     # 누적 (합)
43     # key 를 정의할때 ( $\min(i, j)$ ,  $\max(i, j)$ ) 와 같이 했으므로
44     #  $h_{ij}$  와  $h_{ji}$  는 같은 key 에 대응되고, 이 두 값은 하나의 edge 에 대응되어야한다.
45     # 우리가 다루는 행렬에서는 이 두 값이 같으므로, 두값을 평균을 내어 처리한다.
46
47     for (i, j), w in edges.items():
48         # 저장된 딕셔너리에서 키인 ( $i, j$ ) 와 그때의 행렬원소 ( $h_{ij}$ ) 를 for 문으로 가져온다.
49         G.add_edge(i, j, weight=w)
50         # 그래프에  $i$  번째 노드와  $j$  번째 노드의 간선으로써 행렬원소 ( $h_{ij}$ ) 를 부여한다.
51
52     return G # NetworkX 패키지의 Graph 클래스에 노드와 간선들이 저장되어있는 객체

```

4 main 에너지 계산 함수

위의 두 내용을 이용하여 에너지 계산에 사용

```

1 def GBBD(mol):
2     # 입력 : qiskit 의 molecule info 와 같이 분자의 정보를 담고있는 클래스인 MolecularData
3     # 이후에 추가 설명
4     mol = run_pyscf(mol, run_scf=1, run_fci=0)
5     # 3. 2 차 정량화 Hamiltonian 얻기
6     ham_int = mol.get_molecular_hamiltonian()
7     ham_fci = get_fermion_operator(ham_int)
8     #H = get_sparse_operator(ham_fci, n_qubits=mol.n_qubits)
9     H = get_number_preserving_sparse_operator(
10     fermion_op=ham_fci,
11     num_qubits=mol.n_qubits,
12     num_electrons=mol.n_electrons,          # 필수
13     spin_preserving=True,                  #  $S_z$  고정 (필요 없으면 None)
14     reference_determinant=None,
15     excitation_level=None)
16     print(H)
17     H_real = H.real
18     '''
19     -----
20     메인코드에서 이부분까지는 2(FCI 행렬 생성) 에서 설명
21     H_real :  $i, j$  번째 Slater Determinant 의 헤밀토니안 에대한 내적값으로 기술된 행렬 (=FCI
        ↳ 행렬)
22     # 이를 대각화 하여 계산하는것이 FCI
23     # sparse 행렬로 표현됨.

```

```

24     '''
25     G = sparse_to_graph(H_real)
26     # 3(FCI 행렬 -> 그래프로 변경) 에서 정의한 함수를 통해  $H_{real}$  행렬을  $G$  라는 그래프의 형태
    ↪ 로 인코딩
27     ccs = list(nx.connected_components(G))
28     # nx.connected_components() 함수를 통해  $G$  라는 그래프 객체에서, 연결된 노드들을 묶어서
    ↪ 표현.
29     # Closed loop 라기보다는, 각 노드들간 을 이동할 수 있다면 그는 하나의 연결요소로 택한다.
30     '''
31     여기까지의 과정이 헤밀토니안 행렬을 그래프를 이용해서 Block 짓는 과정.
32     즉, 서로 연관성이 있는 basis(Slater Determinant) 끼리 묶어 부분 행렬을 생성
33     '''
34
35     sub_mat_idx_g = list(ccs[0])
36     # nx.connected_components(G) 를 통해 생성된 묶음들 중에서 0 번 idx,
37     # 즉 제일 큰 묶음을 나타내는 그래프의 인덱스를 저장
38     H_sub_g = H[sub_mat_idx_g, :][:, sub_mat_idx_g]
39     # 제일 큰 묶음에 포함되는 인덱스들만을 이용하여 부분행렬을 생성.
40     print(H_sub_g)
41     # 이를 확인용으로 출력
42     eigval_g, eigvec = eigsh(H_sub_g, k=1, which='SA')
43     # 이 부분행렬을 대각화
44     E_g = eigval_g[0]
45     # 제일 큰 묶음으로 구성된 부분행렬의 가장 작은 고윳값이 바닥상태 에너지에 대응된다.
46     # 이를  $E_g$  로 저장.
47
48     sub_mat_idx_e = list(ccs[1])
49     # 즉 제일 " 두번째로 큰 " 묶음을 나타내는 그래프의 인덱스를 저장
50     H_sub_e = H[sub_mat_idx_e, :][:, sub_mat_idx_e]
51     # 마찬가지로 이를 이용해 부분행렬을 생성
52     print(H_sub_e)
53     # 확인용 출력
54     eigval_e, eigvec = eigsh(H_sub_e, k=1, which='SA')
55     # 이 부분행렬을 대각화
56     E_e = eigval_e[0]
57     # 두번째로 큰 묶음으로 구성된 부분행렬의 가장 작은 고윳값이 1 차 여기 상태 에너지에 대응된
    ↪ 다.
58     # 이를  $E_g$  로 저장.
59
60     return E_g, E_e # 바닥상태 에너지와 1 차 여기상태 에너지를 함수의 결과로 출력.

```

5 예시코드 (H_2 PES)

PES 란 Potential Energy Surface 로 분자의 거리를 변화시켜가며 에너지를 플롯한 그래프를 의미한다. 논문에서 여러 H_2 클러스터에 대해서 이 PES 를 얻은 결과가 있는데, 이를 얻기위한 계산이다.

```
1 #H2
2 H2_energy_arr_g = []
3 H2_energy_arr_e1 = []
4 # GBBD 함수에서는 바닥상태 에너지와 1 차여기상태 에너지를 출력하므로,
5 # 이 두가지를 담을 리스트를 준비한다.
6 for dist in np.arange(0.4,2.5, 0.1):
7     # 단위는 Angstrom 이며, 0.4 부터 2.5 까지 거리를 0.1 만큼 변화시켜가며 에너지를 계산한다.
8     geometry = [('H', (0., 0., 0.)),
9                 ('H', (dist, 0., 0.))]
10    # 위 H2 Cluster 구조에서 알 수 있듯이 두 수소원자는 dist 만큼 떨어져있어야하므로
11    # x 축을 기준으로 dist 만큼 떨어져 있도록 기하학적 구조를 설정
12    basis = 'sto-3g'
13    # toy model 의 계산이므로 basis 는 가장 작은 sto-3g 를 사용
14    mol = MolecularData(geometry, basis, multiplicity=1, charge=0)
15    # 기하학적 구조와 basis, 그리고 H2 분자의 spin 과 전하를 고려하여 파라미터를 주고
16    # MolecularData() 함수를 통해 mol 이라는 객체를 생성
17    energy_g, energy_e1 = GBBD(mol)
18    # 입력 : 분자의 기하학적인 구조를 담고있는 객체 (mol)
19    # 출력 : 바닥상태 에너지, 1 차 여기상태 에너지
20    H2_energy_arr_g.append(energy_g)
21    H2_energy_arr_e1.append(energy_e1)
22    # GBBD 함수의 두 결과를 각각 다른 리스트에 저장
23
```