k-RDM の高速化手法について

計数工学科 3 年 03-220602 浜口広樹

2023 年 2 月 5 日

1　問題設定

kRDM を求めよ。必要な計算量はいくらほどか。

2　行列積

まず、先んじて以下の行列を考えます。

$$\begin{aligned}&\text{jordan\_wigner\_ladder\_sparse}(Q, a, 0)\\ =&\begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix}^{a} \otimes \begin{pmatrix}0 & 1\\ 0 & 0\end{pmatrix} \otimes I_{2^{Q-a-1}}\end{aligned}$$

ただし、$\otimes$ はクロネッカー積を表すとし、累乗に関してもクロネッカー積に関するものです。(書き方の流儀が分かりませんが…)(また、先日この関数の定義が分からないと私が騒いでいましたが、ちゃんと落ち着いて読めば分かりました。申し訳ありません。)

この時、その内 k 個を選んだ際の順列に関する積が必要なので、それを求めます。

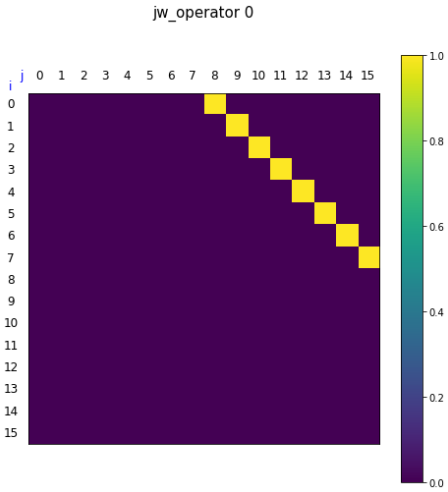

図 1　0 番目

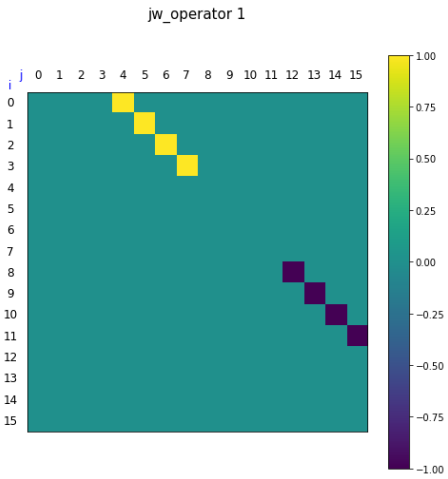

図 2　1 番目

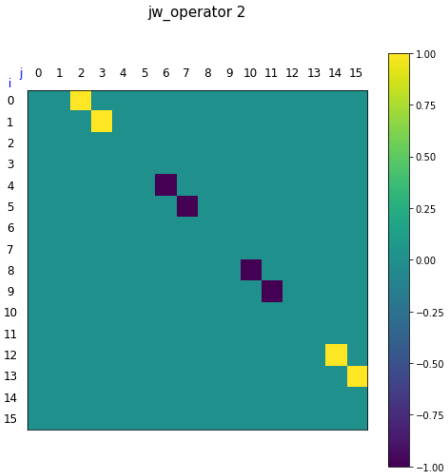

図 3　2 番目

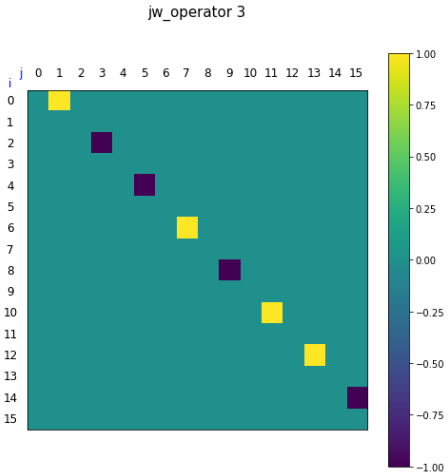

図 4　3 番目

3　行列積の演算

以下で求められます。

ソースコード 1　Python

```python
def _make_jordan_wigners_mul_vec(Q: int, k: int, vec: np.ndarray,
                                 ) -> Dict[int, np.ndarray]:
    assert 1 <= k <= Q

    n_hilbert = 2**Q
    jordan_wigners_mul_vec = dict()

    for ps in combinations(range(Q)[::-1], k):
        data = []
        row = []
        col = []
        offset = sum(1 << (Q-1-p) for p in ps)
        mask = (n_hilbert-1) ^ offset
        r = n_hilbert-offset
        while r > 0:
            r = (r-1) & mask
            data.append(+1 if (sum(bin(r >> (Q-1-p))[2:].count('1
                ')
                                   for p in ps) % 2 == 0) else
                                      -1)
            row.append(r)
            col.append(r+offset)

        ans = csr_matrix((data, (row, col)),
                         shape=(n_hilbert, n_hilbert)) @ vec
        jordan_wigners_mul_vec[_getIdx(Q, *ps)] = ans
        jordan_wigners_mul_vec[_getIdx(Q, *ps[::-1])] =\
            ans*((-1)**((k*(k-1)//2) % 2))

    return jordan_wigners_mul_vec
```

その理由を、大筋だけですが以下に書きます。

関数 $f(i,P):\mathbb{N}\times\mathbb{N}^{\mathbb{N}}\to\pm1$ を以下の様に定義します。

$$f(i,P)=\begin{cases}+1 & (\sum_{p\in P}\mathrm{popcount}\,(i>>p)+\epsilon(P)\equiv 0 \mod 2)\\ -1 & (\text{otherwise})\end{cases}$$

ただし、関数 $\mathrm{popcount}(i):\mathbb{N}\to\mathbb{N}$ は、i を二進数表記した際の 1 の個数 (立っているビット数) とし、$i>>p$ で i を p 桁右シフトした値 ($\lfloor\frac{i}{2^p}\rfloor$) を表すものとします。

また、& は 2 進数表記した際の bit 毎の and を示すものとし、$\mathrm{bitsum}(P) = \sum_{p \in P} 2^p$ とします。

すると、求めるべき行列の i 行 j 列成分に対して、P を掛け合わせる行列の添字列として、

- 上から右への斜め線状に要素があること: $\delta_{i+\mathrm{bitsum}(P),j}$
- 要素が零かどうかの規則: $i\&\mathrm{bitsum}(P) = 0$
- 要素の符号についての規則: $f(i, P)$

という条件で示されることが分かり、これが演算を通して常に成立する不変条件となることが、帰納法により確かめられます。

よって、正当性が示されます。

ちなみに、実装上の工夫として、所謂 3 乗の dp と呼ばれる問題に対するコードを一部活用しています。

4 kRDM

以上の議論を基に、kRDM を求めます。コードを読む方が早い気もするので、以下に載せます。方針は slack にて共有して頂いたものの、ほぼそのまま (なはず) です。

ソースコード 2 kRDM

```
def fast_compute_k_rdm(k: int, vec: np.ndarray,
                       verbose: bool = True) -> np.ndarray:
    Q = int(np.log2(vec.shape[0]))
    assert 2**Q == vec.shape[0]

    # 要請: k <= Q でなければならない(そうでなければ全て0)
    assert 1 <= k <= Q

    QCk = factorial(Q)//factorial(k)//factorial(Q-k)
    QPk = factorial(Q)//factorial(Q-k)

    rdm_data = [0j]*(QPk**2)
    rdm_idx = [0]*(QPk**2)

    fixed_k = _generate_fixed_parity_permutations(k)

    jordan_wigners_mul_vec = _make_jordan_wigners_mul_vec(Q, k,
        vec)

    idx_up = Q**k

    i = 0
    for ps, qs in tqdm(combinations(combinations(range(Q), k),
        2),
                       total=QCk*(QCk-1)//2,
                       disable=not verbose):
```

```
        bra = jordan_wigners_mul_vec[_getIdx(Q, *ps[::-1])]
        ket = jordan_wigners_mul_vec[_getIdx(Q, *qs)]
        val = np.dot(bra.conj(), ket)
        val_conj = val.conj()

        for perm1, parity1 in _generate_parity_permutations(ps,
            fixed_k):
            val_p1 = val*parity1
            val_conj_p1 = val_conj*parity1
            idx1 = _getIdx(Q, *perm1)
            for perm2, parity2 in _generate_parity_permutations(
                qs, fixed_k):
                idx2 = _getIdx(Q, *perm2)
                rdm_idx[i] = idx1*idx_up+idx2
                rdm_data[i] = val_p1*parity2
                i += 1
                rdm_idx[i] = idx2*idx_up+idx1
                rdm_data[i] = val_conj_p1*parity2
                i += 1

    for ps in combinations(range(Q), k):
        bra = jordan_wigners_mul_vec[_getIdx(Q, *ps[::-1])]
        ket = jordan_wigners_mul_vec[_getIdx(Q, *ps)]
        val = np.dot(bra.conj(), ket)
        val_conj = val.conj()
        gpp = _generate_parity_permutations(ps, fixed_k)
        for perm1, parity1 in gpp:
            val_p1 = val*parity1
            idx1 = _getIdx(Q, *perm1)
            for perm2, parity2 in gpp:
                idx2 = _getIdx(Q, *perm2)
                rdm_idx[i] = idx1*idx_up+idx2
                rdm_data[i] = val_p1*parity2
                i += 1

    return coo_matrix((rdm_data, ([0]*len(rdm_data), rdm_idx)),
                      shape=(1, Q**(2*k)), dtype=complex)\
        .toarray()\
        .reshape(tuple(Q for _ in range(2*k)))
```

5 計算量

非零要素数は $({}_QP_k)^2$ 個あり、その内で wedge 積同様の同値類 (「up to sign で等しい」もの) のような考え方をすると、本当に必要な要素は ${}_{{}_QC_k}C_2 + {}_QC_k$ 個です。

前者が上側と下側で添字集合が異なる場合、後者が上側と下側で添字集合が同一の場合を表し

ます。

また、各同値類の代表元を求めるのに必要な計算量は、代表元を添字に関してソートされたものにすれば、転倒数 (物理で言うと $\epsilon(P)$) を求めるのに必要な計算などをスキップできるので、前計算された $c_{j_1}...c_{j_k}|\psi>$(ベクトル) 同士の内積だけで計算出来て、これは 2^Q だけで可能です。

そして、$c_{j_1}...c_{j_k}|\psi>$ の前計算は、先程の議論と、「代表元を添字に関してソートされたもの」にするという話から、$\mathcal{O}\big(2^Q\big)$ で出来ます。(実際はもっと少ない)

以上より、

$$\mathcal{O}\big(2^Q{}_QC_k + 2^Q({}_{Q\mathrm{C}_k}C_2 + {}_QC_k) + ({}_QP_k)^2\big)$$

が計算量となります。第一項が前計算、第二項が代表元の計算、第三項が非零要素を全て求めるのに必要な計算量です。

これまでの計算では前計算に行列積を使うので $(2^Q)^2$ が必要でしたが、それが消えたので、

$$\mathcal{O}\big(2^Q({}_{Q\mathrm{C}_k}C_2 + {}_QC_k) + ({}_QP_k)^2\big)$$

と書くことも出来き、${}_{Q\mathrm{C}_k}C_2 >> {}_QC_k$ なので、

$$\mathcal{O}\big(2^Q({}_{Q\mathrm{C}_k}C_2) + ({}_QP_k)^2\big)$$

と書くことも出来ます。

尤も、実測上は、そこはあまりボトルネックになっていなかったので、高々 50 秒中の 2,3 秒が縮んだ程度です。

6 課題点

この時、$Q = 10, k = 4$ で、上の式は 4.830e+07 という数値を示すのですが、それと照らし合わせると計算の遅さが際立って個人的には非常に悔しいです。

本来、この計算量に対しては (1e7/sec と考えると)4 秒くらいで終わってもおかしくないはずなのですが、私の実装が悪いのか、あるいは素の Python の限界なのか、40 秒くらいは掛かります。

(wedge 積に関しても同様で、演算回数がざっと 1e6/sec という結果になります。私は普段競プロで Python を殆ど使わないので、肌感覚が少しおかしいのかも知れませんが、それにしてもやはり遅そうです)

これに関して、実はここ何日かなり定数倍改善の試行錯誤をしていたのですが、どうにも分からずお手上げ状態に近いです。Cython も試そうとはしましたが、色々なモジュールにこのコードが依存している以上、流石に労力に対するリターンが少ないかなという気でいます。

尤も、wedge 積も含めて、計算機のメモリに載る範囲内では殆どのケースがかなり高速に動作するはずです。

以上です。