关于试验细节的一些说明

- 苏博南
- 可以先看“实验的结果”部分，不需要了解详细过程也可以看到验证的结果。

实验流程的说明

首先，实验项目在https://github.com/SugarSBN/QCNN-robustness-verifier，全部使用C++，不依赖任何第三方库，可直接编译执行。

大体流程是，有一个类$Polygon$，该类有一个重要的成员变量：$vector < PureState > \ points$，它里面是四个PureState，构成了一个四边形。

$Polygon$里还有一个成员变量：$center :: PureState$，表示我们需要验证的量子态。

然后我们对这个Polygon验证，这部分在“域的验证”这一部分说明。

一旦Polygon得到验证，那么就可以通过数值计算得到一个鲁棒域。这一部分在“鲁棒界的确定说明”。

然后是“实验的结果”。

最后讨论了“Polygon”的选择。

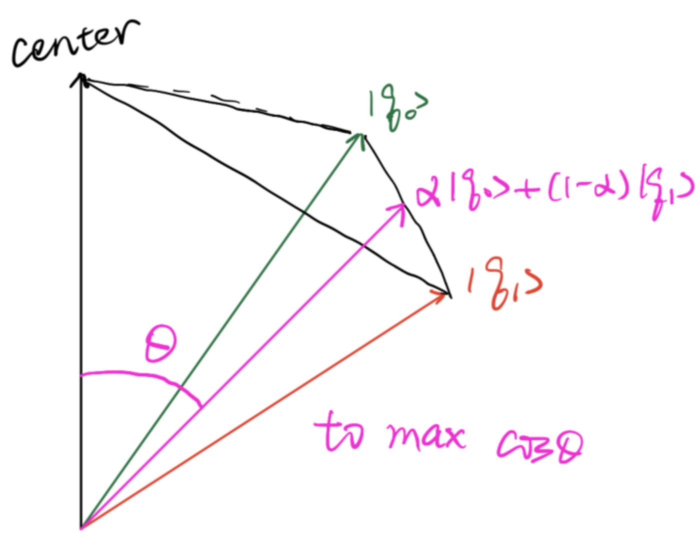
域的验证

```
vector<bool> Polygon :: verify(int ans){
    PureState tmp;
    vector<bool> res;
    for (int i = 0;i < points.size();i++){
        tmp = points[i];
        tmp.apply_circuit(c);
        res.push_back(tmp.predict() == ans);
    }
    return res;
}
```

verify函数非常简单，就是给出一个类ans，然后对polygon的四个顶点分类，返回这四个顶点是否都被分到了ans类里。

鲁棒界的确定

一旦四个顶点都被verify了，就可以开始计算鲁棒域。



我们可以先考虑下对于四边形的一条边q_0q_1，和center组成的四面体。（当然这是为了直观理解，高维空间并不好定义几何体）

可以确定的是，我们需要做的事情，就是寻找：

$$max_{\alpha\in[0,1]}\frac{center\cdot(\alpha|q_0\rangle+(1-\alpha)|q_1\rangle)}{||\alpha|q_0\rangle+(1-\alpha)|q_1\rangle||}=cos\theta$$

因为我们需要寻找最小的θ。如果对于这条边，找到了最小的θ，再对剩下三条边都找一下，然后取最小值，就可以通过之前的理论推导得到一个ε。

```
double Polygon :: robust_bound() const{
    double res = 0;
    for (int i = 0;i < points.size() - 1;i++)    res = max(res, cal(i, i + 1));
    res = max(res, cal(0, points.size() - 1));
    return 1 - (1 + ((1 << nqubits) - 1) * res) / (1 << nqubits);
}

double Polygon :: cal(int i, int j) const{
    double costheta1 = ((1 << nqubits) * center_state.fidelity(points[i]) - 1) / ((1 << nqubits) - 1);
    double costheta2 = ((1 << nqubits) * center_state.fidelity(points[j]) - 1) / ((1 << nqubits) - 1);

    return max(costheta1, costheta2) * sqrt(2);
}
```

这部分的数值计算有些费解。首先，在基$\{X,Y,Z,I\}^{\otimes n}-I^{\otimes n}$这个基下，是不满足纯态的$|\vec{v}|=1$的，我们需要对基做归一化。

其实归一化后的基是$\sqrt{2^n-1}(\{X,Y,Z,I\}^{\otimes n}-I^{\otimes n})$。前面系数是怎么得来的？可以看一下简单推导：

$$
\begin{gathered}
\text{原始情况下：} \\
\rho=\frac{1}{2^n}(I+\vec{v}\cdot\vec{\sigma}) \\
Tr(\sigma_i\sigma_j)=2^n\delta_{ij} \\
\text{然后就有：} \\
Tr(\rho^2)=\frac{1}{4^n}(2^n+|\vec{v}|^2 2^n) \\
\text{其中}|\vec{v}|^2\text{乘的}2^n\text{就是}Tr(\sigma_i\sigma_j)=2^n\delta_{ij}\text{前的那个}2^n. \\
\text{因为我们想要}Tr(\rho^2)=1\Leftrightarrow|\vec{v}|^2=1 \\
\text{所以就要使得}Tr(\sigma_i\sigma_j)=(4^n-2^n)\delta_{ij} \\
\text{所以基前面需要乘一个系数}\sqrt{2^n-1}
\end{gathered}
$$

然后此时有：

$$
\begin{aligned}
Fidelity(\psi_1,\psi_2)&=|\langle\psi_1|\psi_2|\rangle|^2 \\
&=\frac{1+(2^n-1)\vec{v_1}\cdot\vec{v_2}}{2^n}
\end{aligned}
$$

回到之前需要最大化的值：

$$
max_{\alpha\in[0,1]}\frac{center\cdot(\alpha|q_0\rangle+(1-\alpha)|q_1\rangle)}{||\alpha|q_0\rangle+(1-\alpha)|q_1\rangle||}=cos\theta
$$

由于我懒得求导，会发现分母$||\alpha|q_0\rangle+(1-\alpha)|q_1\rangle||\geq\frac{1}{\sqrt{2}}$的（因为转的比较小，$|q_0\rangle$和$|q_1\rangle$成锐角），所以我直接把上面的结果算作：

$$
\sqrt{2}*max(cos(center,|q_0\rangle),cos(center,|q_1\rangle))
$$

也就有了cal()函数。

然后robust_bound函数就是对每条边都cal了一下，最后取个最大的$cos\theta$，最后计算出bound：

$$
\varepsilon=1-maxFidelity=1-max\frac{1+(2^n-1)cos\theta}{2^n}
$$

实验的结果

前面都是原理性说明，我不是很有信心表达清楚。但是我们可以看一下结果。

```
bool Verifier :: random_sampling_check(int samples) const{
    srand((int) time (NULL));
    PureState nq = q;

    printf("Robust bound: %.4lf\n", robust_bound);
    for (int t = 0;t < samples;t++){

        double phi = (double)rand() / (double)RAND_MAX * 7;
        int tg = rand() % nqubits;
        int gate = rand() % 3;
        double distance;
        Gate g = gate == 0 ? RX : (gate == 1 ? RY : RZ);
        nq.apply_operator(Operator(g, tg, vector<int>{}, phi));
        distance = 1 - q.fidelity(nq);

        while(1 - q.fidelity(nq) > robust_bound || distance < (robust_bound - 0.01)){

            phi = (double)rand() / (double)RAND_MAX * 7;
            tg = rand() % nqubits;
            gate = rand() % 3;
            g = gate == 0 ? RX : (gate == 1 ? RY : RZ);
            nq = PureState(q);
            nq.apply_operator(Operator(g, tg, vector<int>{}, phi));
            distance = 1 - q.fidelity(nq);
        }
        nq.apply_circuit(c);
        if (predict != nq.predict())
            printf("sample %d: Distance = %.4lf CENTER = %d\n SAMPLE = %d\n", t, distance, predict, nq.predict());
        else printf("sample %d: Distance = %.4lf CENTER = SAMPLE = %d\n", t, distance, predict);
    }
    return 1;
}
```

可以看一下我取点的随机策略，我随机生成了一个角度phi，一个target qubit，一个gate(RX, RY, RZ)。然后将gate作用在target qubit上转phi角度。

为了增加数据的强度，如果产生的状态和center态的distance离得太远（差0.01以上），则我会重新生成。也就是说，sample的点一定会特别靠近验证出来的robust边界。

```
0:
Guan's robust bound: 0.016063
Robust bound: 0.1397
sample 0: Distance = 0.1307 CENTER = SAMPLE = 0
sample 1: Distance = 0.1380 CENTER = SAMPLE = 0
sample 2: Distance = 0.1304 CENTER = SAMPLE = 0
sample 3: Distance = 0.1386 CENTER = SAMPLE = 0
sample 4: Distance = 0.1334 CENTER = SAMPLE = 0
sample 5: Distance = 0.1307 CENTER = SAMPLE = 0
sample 6: Distance = 0.1371 CENTER = SAMPLE = 0
sample 7: Distance = 0.1341 CENTER = SAMPLE = 0
sample 8: Distance = 0.1329 CENTER = SAMPLE = 0
sample 9: Distance = 0.1394 CENTER = SAMPLE = 0
sample 10: Distance = 0.1365 CENTER = SAMPLE = 0
sample 11: Distance = 0.1331 CENTER = SAMPLE = 0
sample 12: Distance = 0.1390 CENTER = SAMPLE = 0
sample 13: Distance = 0.1335 CENTER = SAMPLE = 0
sample 14: Distance = 0.1299 CENTER = SAMPLE = 0
sample 15: Distance = 0.1340 CENTER = SAMPLE = 0
sample 16: Distance = 0.1359 CENTER = SAMPLE = 0
sample 17: Distance = 0.1382 CENTER = SAMPLE = 0
sample 18: Distance = 0.1322 CENTER = SAMPLE = 0
sample 19: Distance = 0.1348 CENTER = SAMPLE = 0
sample 20: Distance = 0.1381 CENTER = SAMPLE = 0
sample 21: Distance = 0.1367 CENTER = SAMPLE = 0
sample 22: Distance = 0.1303 CENTER = SAMPLE = 0
sample 23: Distance = 0.1328 CENTER = SAMPLE = 0
sample 24: Distance = 0.1309 CENTER = SAMPLE = 0
sample 25: Distance = 0.1326 CENTER = SAMPLE = 0
sample 26: Distance = 0.1357 CENTER = SAMPLE = 0
sample 27: Distance = 0.1370 CENTER = SAMPLE = 0
sample 28: Distance = 0.1327 CENTER = SAMPLE = 0
```

```
sample 74: Distance = 0.1389 CENTER = SAMPLE = 0
sample 75: Distance = 0.1370 CENTER = SAMPLE = 0
sample 76: Distance = 0.1387 CENTER = SAMPLE = 0
sample 77: Distance = 0.1370 CENTER = SAMPLE = 0
sample 78: Distance = 0.1389 CENTER = SAMPLE = 0
sample 79: Distance = 0.1344 CENTER = SAMPLE = 0
sample 80: Distance = 0.1389 CENTER = SAMPLE = 0
sample 81: Distance = 0.1309 CENTER = SAMPLE = 0
sample 82: Distance = 0.1320 CENTER = SAMPLE = 0
sample 83: Distance = 0.1375 CENTER = SAMPLE = 0
sample 84: Distance = 0.1334 CENTER = SAMPLE = 0
sample 85: Distance = 0.1353 CENTER = SAMPLE = 0
sample 86: Distance = 0.1307 CENTER = SAMPLE = 0
sample 87: Distance = 0.1359 CENTER = SAMPLE = 0
sample 88: Distance = 0.1315 CENTER = SAMPLE = 0
sample 89: Distance = 0.1376 CENTER = SAMPLE = 0
sample 90: Distance = 0.1345 CENTER = SAMPLE = 0
sample 91: Distance = 0.1372 CENTER = SAMPLE = 0
sample 92: Distance = 0.1315 CENTER = SAMPLE = 0
sample 93: Distance = 0.1363 CENTER = SAMPLE = 0
sample 94: Distance = 0.1310 CENTER = SAMPLE = 0
sample 95: Distance = 0.1321 CENTER = SAMPLE = 0
sample 96: Distance = 0.1347 CENTER = SAMPLE = 0
sample 97: Distance = 0.1299 CENTER = SAMPLE = 0
sample 98: Distance = 0.1380 CENTER = SAMPLE = 0
sample 99: Distance = 0.1363 CENTER = SAMPLE = 0
Verified
```

可以看到，对第一个center态，我随即抽取了100个sample，都得到了验证。而且数据中不乏离robust bound特别近的点（distance = 0.1394等）

而我也输出了官老师lemma1的robust bound，对于第一个样本点guan老师的robust bound只给了0.016。

如果不输出详细的sample过程，我们可以对不同的center态进行random sampling check。我选择了MNIST数据集前一百个测试点，每个测试点random sample 一百次。共耗时将近1小时，结果如下：

```
0:
Guan's robust bound: 0.016063
Robust bound: 0.1397
Verified
1:
Guan's robust bound: 0.056402
Robust bound: 0.1397
Verified
2:
Guan's robust bound: 0.159436
Robust bound: 0.2445
Verified
3:
Guan's robust bound: 0.180515
Robust bound: 0.0715
Verified
4:
Guan's robust bound: 0.178353
Robust bound: 0.0383
Verified
5:
Guan's robust bound: 0.013586
Robust bound: -0.2229
Verified
6:
Guan's robust bound: 0.048491
Robust bound: 0.1397
Verified
7:
Guan's robust bound: 0.020605
Robust bound: -0.1168
Verified
```

```
8:
Guan's robust bound: 0.002521
Robust bound: -0.3260
Verified
9:
Guan's robust bound: 0.003122
Robust bound: -0.1981
Verified
10:
Guan's robust bound: 0.196769
Robust bound: 0.0383
Verified
11:
Guan's robust bound: 0.029245
Robust bound: 0.0715
Verified
12:
Guan's robust bound: 0.036707
Robust bound: 0.1397
Verified
13:
Guan's robust bound: 0.077620
Robust bound: 0.2093
Verified
14:
Guan's robust bound: 0.006968
Robust bound: -0.0875
Verified
15:
Guan's robust bound: 0.126743
Robust bound: 0.2445
Verified
```

```
16:
Guan's robust bound: 0.020567
Robust bound: -0.1722
Verified
17:
Guan's robust bound: 0.006098
Robust bound: -0.1722
Verified
18:
Guan's robust bound: 0.084103
Robust bound: 0.1743
Verified
19:
Guan's robust bound: 0.241075
Robust bound: 0.2093
Verified
20:
Guan's robust bound: 0.009538
Robust bound: -0.2463
Verified
21:
Guan's robust bound: 0.071703
Robust bound: 0.1397
Verified
22:
Guan's robust bound: 0.110057
Robust bound: 0.2093
Verified
23:
Guan's robust bound: 0.003046
Robust bound: -0.2891
Verified
24:
Guan's robust bound: 0.046507
Robust bound: 0.0715
Verified
```

```
25:
Guan's robust bound: 0.005168
Robust bound: -0.2229
Verified
26:
Guan's robust bound: 0.078457
Robust bound: 0.2093
Verified
27:
Guan's robust bound: 0.222045
Robust bound: 0.2093
Verified
28:
Guan's robust bound: 0.142321
Robust bound: 0.2445
Verified
29:
Guan's robust bound: 0.097727
Robust bound: 0.2093
Verified
30:
Guan's robust bound: 0.027872
Robust bound: 0.1054
Verified
31:
Guan's robust bound: 0.214733
Robust bound: 0.1743
Verified
32:
Guan's robust bound: 0.057759
Robust bound: 0.0057
Verified
```

```
33:
Guan's robust bound: 0.050141
Robust bound: 0.1743
Verified
34:
Guan's robust bound: 0.038758
Robust bound: 0.1397
Verified
35:
Guan's robust bound: 0.012102
Robust bound: -0.1450
Verified
36:
Guan's robust bound: 0.053520
Robust bound: 0.1743
Verified
37:
Guan's robust bound: 0.188851
Robust bound: 0.2445
Verified
38:
Guan's robust bound: 0.176227
Robust bound: 0.1397
Verified
39:
Guan's robust bound: 0.012327
Robust bound: 0.2093
Verified
40:
Guan's robust bound: 0.180572
Robust bound: 0.2445
Verified
```

```
41:
Guan's robust bound: 0.031219
Robust bound: 0.0057
Verified
42:
Guan's robust bound: 0.001362
Robust bound: -0.3083
Verified
43:
Guan's robust bound: 0.167425
Robust bound: 0.0383
Verified
44:
Guan's robust bound: 0.113009
Robust bound: 0.2445
Verified
45:
Guan's robust bound: 0.184837
Robust bound: 0.2093
Verified
46:
Guan's robust bound: 0.027377
Robust bound: 0.0715
Verified
47:
Guan's robust bound: 0.183504
Robust bound: 0.2445
Verified
48:
Guan's robust bound: 0.046016
Robust bound: 0.2093
Verified
```

```
49:
Guan's robust bound: 0.046123
Robust bound: 0.0383
Verified
50:
Guan's robust bound: 0.041922
Robust bound: -0.0573
Verified
51:
Guan's robust bound: 0.201275
Robust bound: 0.2093
Verified
52:
Guan's robust bound: 0.042745
Robust bound: 0.0715
Verified
53:
Guan's robust bound: 0.022952
Robust bound: 0.1743
Verified
54:
Guan's robust bound: 0.169527
Robust bound: 0.2093
Verified
55:
Guan's robust bound: 0.008077
Robust bound: 0.0383
Verified
56:
Guan's robust bound: 0.042894
Robust bound: 0.1397
Verified
```

```
57:
Guan's robust bound: 0.035769
Robust bound: -0.0573
Verified
58:
Guan's robust bound: 0.159805
Robust bound: 0.2445
Verified
59:
Guan's robust bound: 0.000065
Robust bound: -0.4117
Verified
60:
Guan's robust bound: 0.018891
Robust bound: -0.1168
Verified
61:
Guan's robust bound: 0.021635
Robust bound: -0.1722
Verified
62:
Guan's robust bound: 0.122760
Robust bound: 0.2093
Verified
63:
Guan's robust bound: 0.036610
Robust bound: 0.0383
Verified
```

```
64:
Guan's robust bound: 0.164429
Robust bound: 0.2093
Verified
65:
Guan's robust bound: 0.026302
Robust bound: 0.0715
Verified
66:
Guan's robust bound: 0.135871
Robust bound: 0.1054
Verified
67:
Guan's robust bound: 0.169062
Robust bound: 0.2093
Verified
68:
Guan's robust bound: 0.183024
Robust bound: 0.2093
Verified
69:
Guan's robust bound: 0.061449
Robust bound: 0.2445
Verified
70:
Guan's robust bound: 0.141673
Robust bound: 0.2093
Verified
71:
Guan's robust bound: 0.022333
Robust bound: 0.0715
Verified
```

```
72:
Guan's robust bound: 0.045174
Robust bound: 0.1397
Verified
73:
Guan's robust bound: 0.015196
Robust bound: 0.1397
Verified
74:
Guan's robust bound: 0.098806
Robust bound: 0.0383
Verified
75:
Guan's robust bound: 0.199000
Robust bound: 0.1054
Verified
76:
Guan's robust bound: 0.102358
Robust bound: 0.2445
Verified
77:
Guan's robust bound: 0.006484
Robust bound: -0.1168
Verified
78:
Guan's robust bound: 0.025671
Robust bound: 0.1054
Verified
79:
Guan's robust bound: 0.098363
Robust bound: 0.2093
Verified
```

```
81:
Guan's robust bound: 0.053861
Robust bound: 0.0383
Verified
82:
Guan's robust bound: 0.112099
Robust bound: 0.2445
Verified
83:
Guan's robust bound: 0.141949
Robust bound: 0.1743
Verified
84:
Guan's robust bound: 0.036890
Robust bound: 0.0715
Verified
85:
Guan's robust bound: 0.213896
Robust bound: 0.2445
Verified
86:
Guan's robust bound: 0.210418
Robust bound: 0.2445
Verified
87:
Guan's robust bound: 0.107412
Robust bound: 0.1397
Verified
88:
Guan's robust bound: 0.059543
Robust bound: 0.0715
Verified
```

```
89:
Guan's robust bound: 0.223104
Robust bound: 0.1743
Verified
90:
Guan's robust bound: 0.008955
Robust bound: 0.1054
Verified
91:
Guan's robust bound: 0.141783
Robust bound: 0.1397
Verified
92:
Guan's robust bound: 0.137643
Robust bound: 0.0383
Verified
93:
Guan's robust bound: 0.110341
Robust bound: 0.1743
Verified
94:
Guan's robust bound: 0.012871
Robust bound: 0.0383
Verified
95:
Guan's robust bound: 0.055387
Robust bound: 0.1743
Verified
96:
Guan's robust bound: 0.053934
Robust bound: 0.1397
Verified
97:
Guan's robust bound: 0.028178
Robust bound: 0.1743
Verified
```

```
97:
Guan's robust bound: 0.028178
Robust bound: 0.1743
Verified
98:
Guan's robust bound: 0.061475
Robust bound: 0.0715
Verified
99:
Guan's robust bound: 0.048472
Robust bound: 0.0383
Verified
```

我觉得结果出人意料得不错，很多时候比官老师的lemma1给出的quick check效果要好很多，而且理论速度更快。

这里可能有疑问，为什么有时候验证出来robust bound是负数？这其实很显然，因为之前求max$cos\theta$时，我直接放缩到$\sqrt{2}$，可能会过大了。

其次

```
void Verifier :: work(){
    poly = Polygon(q, nqubits, c, poly_param);
    vector<bool> veri_res = poly.verify(predict);
    bool not_verified = 0;
    for (int i = 0;i < veri_res.size();i++) if (veri_res[i] == 0)   not_verified = 1;
    while(not_verified){
        poly_param -= 0.05;
        poly = Polygon(q, nqubits, c, poly_param);
        vector<bool> veri_res = poly.verify(predict);
        not_verified = 0;
        for (int i = 0;i < veri_res.size();i++) if (veri_res[i] == 0)   not_verified = 1;
    }
    robust_bound = poly.robust_bound();
}
```

如果对于一个eps，我们的polygon四个顶点并没有被完全分入同一个类怎么办？为了节省时间，我使用了最简单朴素的方法。

即我把初始eps设成1.6，然后如果没被验证，就把eps - 0.05，缩小四边形直到能验证了为止。因此会发现验证出来的robust bound都是一些离散的值。

当然这是为了节省时间，也可以缩短步长或者做更精细的启发式验证。

Polygon的选择

```
Polygon :: Polygon(PureState center, int nnqubits, Circuit nc, double eps){
    points.clear();
    nqubits = nnqubits;
    c = nc;

    center_state = center;
    center.apply_operator(Operator(RY, 0, vector<int>{}, eps));
    points.push_back(center);

    center = center_state;
    center.apply_operator(Operator(RY, 1, vector<int>{}, eps));
    points.push_back(center);

    center = center_state;
    center.apply_operator(Operator(RY, 0, vector<int>{}, -eps));
    points.push_back(center);

    center = center_state;
    center.apply_operator(Operator(RY, 1, vector<int>{}, -eps));
    points.push_back(center);
}
```

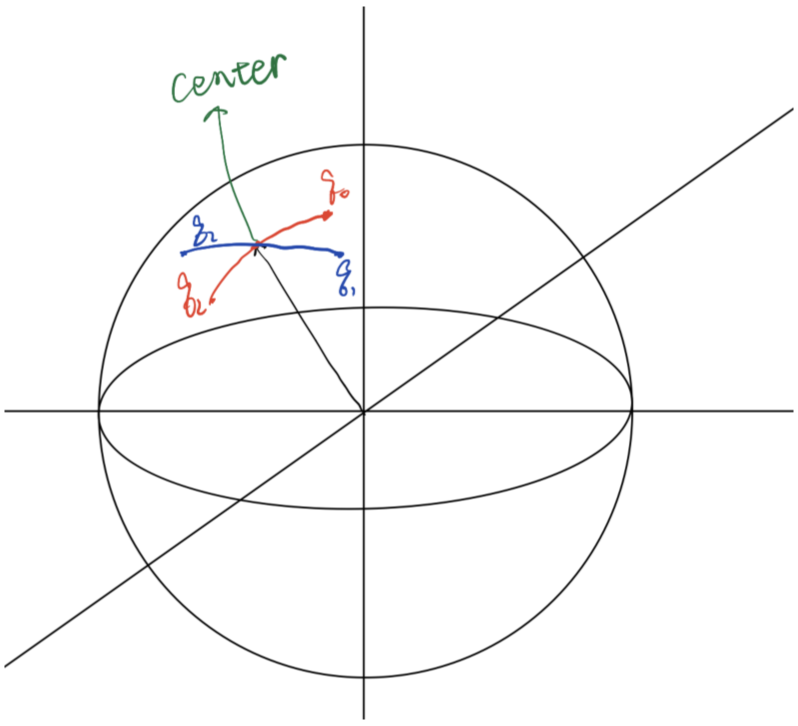
这部分，center是一个PureState，然后我选取了一个四边形。即四个量子态：

- 将center的第0个qubit作用门$RY(eps)$得到的量子态$|q_0\rangle$
- 将center的第1个qubit作用门$RY(eps)$得到的量子态$|q_1\rangle$
- 将center的第0个qubit作用门$RY(-eps)$得到的量子态$|q_2\rangle$
- 将center的第1个qubit作用门$RY(-eps)$得到的量子态$|q_3\rangle$

为什么这么取呢？实际上，将量子态作用门$I \otimes I \otimes ... \otimes I \otimes RY$并不对应着绕着轴$I \otimes I \otimes ... \otimes I \otimes RY$进行旋转？（这里我不清楚，我就认为他不对）

但是可以换一种理解方式，因为作用门后量子态还是纯态，所以一定是绕着某个轴旋转的。（考虑之前推导的$|\psi\rangle$对应的向量$\overrightarrow{v}$）

因为高维空间的旋转并不是很好理解，可以再换一种想法。把量子态既然看作向量了，由于我们取的是旋转一个负角度，作用门的本质是向量中一些分量变大，一些分量减小；而作用负数RY正好可以使得原来减小的分量变大，原来变大的分量减小。所以实际上，center量子态一定在$|q_0\rangle$和$|q_2\rangle$的连线上。



想法就是虽然我不知道是绕着哪个轴旋转的，但它一定是个旋转。数字上，就是将向量一些分量变大一些分量减小。由于对称性，center态一定是被$|q_0\rangle, ...|q_3\rangle$包围的，这就足够了。