

```

1  #include <iostream>
2  #include "QSolver_linear_solver_test.h"
3
4  QCircuit amplitude_encode(qvec q, vector<double> data)
5  {
6      if (data.size() > (1 << q.size()))
7      {
8          throw exception("error");
9      }
10     while(data.size() < (1 << q.size()))
11     {
12         data.push_back(0);
13     }
14     QCircuit qcir;
15     double sum_0 = 0;
16     double sum_1 = 0;
17     size_t high_bit = (size_t)log2(data.size())-1;
18     if (high_bit == 0)
19     {
20         if ((data[0] + data[1]) > 1e-20)
21         {
22             qcir << RY(q[0], 2 * acos(data[0] / sqrt(data[0] * data[0] + data[1] *
23                 data[1])));
24         }
25     }
26     else
27     {
28         for (auto i = 0; i < (data.size() >> 1); i++)
29         {
30             sum_0 += data[i] * data[i];
31             sum_1 += data[i + (data.size() >> 1)] * data[i + (data.size() >> 1)];
32         }
33         if (sum_0+sum_1 > 1e-20)
34             qcir << RY(q[high_bit], 2 * acos(sqrt(sum_0 / (sum_0+sum_1))));
35         else
36         {
37             throw exception("error");
38         }
39         if (sum_0 > 1e-20)
40         {
41             qvec temp({ q[high_bit] });
42             vector<double> vtemp(data.begin(), data.begin() + (data.size() >> 1));
43             qcir <<X(q[high_bit])<< amplitude_encode(q - temp, vtemp).control({ q
44                 [high_bit] }) << X(q[high_bit]);
45         }
46         if (sum_1 > 1e-20)
47         {
48             qvec temp({ q[high_bit] });
49             vector<double> vtemp(data.begin() + (data.size() >> 1), data.end());
50             qcir << amplitude_encode(q - temp, vtemp).control({ q[high_bit] });
51         }
52     }
53 }

```

```

52     return qcir;
53 }
54
55 QCircuit init_superposition_state(qvec q, size_t d)
56 {
57     QCircuit qcir;
58
59     size_t highest_bit = (size_t)log2(d-1);
60
61     if (d == (1 << (int)log2(d)))
62     {
63         for (auto i = 0; i < (int)log2(d); i++)
64         {
65             qcir << H(q[i]);
66         }
67     }
68     else if (d == 3)
69     {
70         qcir << RY(q[1], 2 * acos(sqrt(2.0 / 3))) << X(q[1]);
71         qcir << H(q[0]).control({ q[1] }) << X(q[1]);
72     }
73     else
74     {
75         qcir << RY(q[highest_bit], 2 * acos(sqrt((1 << highest_bit)*1.0 / d)));
76         QCircuit qcir1;
77         for (auto i = 0; i < highest_bit; i++)
78         {
79             qcir1 << H(q[i]);
80         }
81         qcir << X(q[highest_bit]) << qcir1.control({ q[highest_bit] }) << X(q
82             [highest_bit]);
83
84         size_t d1 = d - (1 << highest_bit);
85         if(d>1)
86         {
87             QCircuit qcir2 = init_superposition_state(q, d1);
88             qcir2.setControl({ q[highest_bit] });
89             qcir << qcir2;
90         }
91     return qcir;
92 }
93
94 QSolver::QSolver(std::string cfg_file):
95     m_grid_number(16), m_cfg_file(cfg_file), m_sparse_coef(20), Cheby_times
96     (128), m_b(500), m_solution(64, 1.0)
97 {
98     for (auto i = 0; i < 64; i++)
99     {
100         //rho
101         if (i % 4 == 0)
102         {
103             m_solution[i] = 1.22531;

```

```

103     }
104     //u
105     else if (i % 4 == 1)
106     {
107         m_solution[i] = 170.104;
108     }
109     //v
110     else if (i % 4 == 2)
111     {
112         m_solution[i] = 0;
113     }
114     //E
115     else if (i % 4 == 3)
116     {
117         m_solution[i] = 221149;
118     }
119 }
120 }
121 QSolver::QSolver(size_t grid_number) :m_grid_number(grid_number),
122                                         Cheby_times(64),m_b(250),m_sparse_coef
123                                         (4),m_sparse_matrix(4*16,0.000001),
124                                         m_residual(16,1),m_solution(16,1)
125 {
126     //coefficient of Chebyshev polynomial
127     m_alpha = { 1.949549963644177, -1.8488512882814025, 1.7487543978311608,
128                -1.6496525062919445,
129                1.5519270299129846, -1.455943195687338, 1.3620459665535882,
130                -1.2705563586796345,
131                1.1817682156051097, -1.095945491847383, 1.0133200852492674,
132                -0.9340902433058914,
133                0.8584195544185835, -0.7864365209351393, 0.718234697381761,
134                -0.653873364863425,
135                0.5933787015467717, -0.5367453997184383, 0.48393867233793647,
136                -0.43489658640845813,
137                0.38953265692367894, -0.3477386335979911, 0.3093874129600493,
138                -0.274336010541416,
139                0.2424285316221708, -0.21349908406872772, 0.18737458295107148,
140                -0.1638774035777065,
141                0.14282784705571605, -0.12404639019677266, 0.10735569929006719,
142                -0.09258239472103647,
143                0.07955856042991463, -0.06812299861332467, 0.05812223575125098,
144                -0.049411290903158406,
145                0.04185422121218744, -0.03532446267101167, 0.02970498645435139,
146                -0.024888292554356185,
147                0.020776263132320862, -0.01727989800452769, 0.014318954104793266,
148                -0.01182150971054103,
149                0.009723472783751415, -0.007968051061520803, 0.00650519962632839,
150                -0.005291059678290213,
151                0.004287400195696609, -0.003461072169953997, 0.0027834831888454113,
152                -0.002230098358538782,
153                0.0017799719295396406, -0.0014153125440208496, 0.0011210837618425746,
154                -0.0008846404522002947,
155                0.0006954007529361845, -0.0005445525905065386, 0.00042479320707228724,

```

```
-0.00033009974110103914,  
141      0.0002555286366486786, -0.00019704149590170955, 0.00015135492154600306, 7  
      -0.00011581190277550591 };  
142  
143      for (auto i = 0; i < m_alpha.size(); i++)  
144      {  
145  
146          m_alpha[i] = sqrt(abs(m_alpha[i]));  
147  
148      }  
149      //init none zero block  
150      for(auto i=0;i<4;i++)  
151          m_none_zero_block.push_back({ 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 });  
152      for(auto i=4;i<8;i++)  
153          m_none_zero_block.push_back({ 4,5,6,7,0,1,2,3,12,13,14,15,8,9,10,11 });  
154      for(auto i=8;i<12;i++)  
155          m_none_zero_block.push_back({ 8,9,10,11,0,1,2,3,12,13,14,15,4,5,6,7 });  
156      for(auto i=12;i<16;i++)  
157          m_none_zero_block.push_back({ 12,13,14,15, 4,5,6,7,8,9,10,11,0, 1,2,3 });  
158      vvj.push_back({ 0});  
159      vvj.push_back({ 1 });  
160      vvj.push_back({ 2 });  
161      vvj.push_back({ 3 });  
162      //init sparse matrix  
163      for (auto i = 0; i < 4; i++)  
164      {  
165          m_sparse_matrix[5*i] = 0.2;  
166      }  
167      for (auto i = 0; i < 4; i++)  
168      {  
169          m_sparse_matrix[16+5*i] = 0.2;  
170      }  
171      for (auto i = 0; i < 4; i++)  
172      {  
173          m_sparse_matrix[32+5*i] = 0.1;  
174      }  
175      for (auto i = 0; i < 4; i++)  
176      {  
177          m_sparse_matrix[48 + 5*i] = 0.1;  
178      }  
179  }  
180  
181  void QSolver::run()  
182  {  
183      auto qm = initQuantumMachine(CPU_SINGLE_THREAD_WITH_ORACLE);  
184      Configuration config = { 10000,10000 };  
185      qm->setConfig(config);  
186      ModuleContext::setContext(qm);  
187      size_t qnum = ceil(log2(m_grid_number))+2;  
188      size_t qtnum = ceil(log2(Cheby_times));  
189      qvec qi(qnum);  
190      qvec qi_anc(1);  
191      qvec qj(qnum);
```

```

192     qvec qj_anc(1);
193     qvec qt(qtnum);
194     qvec qlist = qi + qi_anc+qj + qj_anc+qt ;
195     QProg prog;
196
197     prog << one_iteration_qcir(qt, qi, qi_anc, qj, qj_anc);
198     auto temp = dynamic_cast<IdealMachineInterface *>(qm);
199     std::string s = transformQProgToOriginIR(prog, qm);
200     std::cout << s << std::endl;
201     ofstream outfile;
202     outfile.open("QuantumLinearSolver.txt", ios_base::out | ios_base::trunc);
203     for (int i = 0; i < s.size(); i++)
204     {
205         stringstream ss;
206         ss << s[i];
207         outfile << ss.str();
208     }
209     outfile.close();
210
211     qm->directlyRun(prog);
212     auto target_state = qm->getQState();
213     vector<qcomplex_t> target;
214     for (auto i = 0; i < 1<<qnum; i++)
215     {
216         target.push_back(target_state[i]);
217     }
218     double sum = 0;
219     for (auto i = 0; i < target.size(); i++)
220     {
221         sum += target[i].real();
222         cout << i << " " << target[i] << endl;
223     }
224     cout << "sum1 " << sum << endl;
225     return;
226 }
227
228 QCircuit QSolver::T_circuit(qvec qi, qvec qi_anc, qvec qj, qvec qj_anc)
229 {
230     QCircuit qcir;
231     size_t qtemp = (size_t)log(m_sparse_coef)+1;
232     string Tname = "truncation_" + to_string((qj + qi_anc + qj_anc).get().size()
233     ());
234     auto trun_oracle = oracle((qj + qi_anc + qj_anc).get(), Tname);
235     qcir << trun_oracle;
236     qcir << init_superposition_state(qj, m_sparse_coef);
237     string OLname = "OL_" + to_string(2 * qi.get().size());
238     auto OL_oracle = oracle((qi + qj).get(), OLname, m_none_zero_block);
239     qcir << OL_oracle;
240     string OMname = "OM_" + to_string(2 * qi.get().size() + 1);
241     auto OM_oracle = oracle((qi + qj+qj_anc).get(), OMname, m_sparse_matrix,
242     vvj);
243     qcir << OM_oracle;
244     return qcir;

```

```

243 }
244
245 QCircuit QSolver::T_circuitv1(qvec qi, qvec qj, qvec qj_anc)
246 {
247     QCircuit qcir;
248     size_t qtemp = (size_t)log(m_sparse_coef) + 1;
249     qcir << init_superposition_state(qj, m_sparse_coef);
250     string OLname = "OL_" + to_string(2 * qi.get().size());
251     auto OL_oracle = oracle((qi + qj).get(), OLname, m_none_zero_block);
252     qcir << OL_oracle;
253     string OMname = "OM_" + to_string(2 * qi.get().size() + 1);
254     auto OM_oracle = oracle((qi + qj + qj_anc).get(), OMname, m_sparse_matrix,
255                             vvj);
256     qcir << OM_oracle;
257     return qcir;
258 }
259
260 QCircuit QSolver::T_circuitv2(qvec qi, qvec qi_anc, qvec qj, qvec qj_anc)
261 {
262     QCircuit qcir;
263     qcir << X(qi_anc[0]);
264     QCircuit qcirl = T_circuitv1(qi, qj, qj_anc);
265     qcirl.setControl({ qi_anc[0] });
266     qcir << qcirl;
267     qcir << X(qi_anc[0]);
268     qcir << X(qj_anc[0]).control({ qi_anc[0] });
269     return qcir;
270 }
271
272 QCircuit QSolver::W_circuit(qvec qi, qvec qi_anc, qvec qj, qvec qj_anc)
273 {
274     QCircuit qcir;
275     qcir << T_circuitv2(qi, qi_anc, qj, qj_anc).dagger();
276     for (auto i = 0; i < qj.size(); i++)
277     {
278         qcir << X(qj[i]);
279     }
280     qcir << X(qj_anc[0]);
281     qcir << Z(qj_anc[0]).control(qj.get());
282     for (auto i = 0; i < qj.size(); i++)
283     {
284         qcir << X(qj[i]);
285     }
286     qcir << X(qj_anc[0]);
287     qcir << Z(qj_anc[0]) << X(qj_anc[0]) << Z(qj_anc[0]) << X(qj_anc[0]);
288     qcir << T_circuitv2(qi, qi_anc, qj, qj_anc);
289
290     qcir << CNOT(qi_anc[0], qj_anc[0]) << CNOT(qj_anc[0], qi_anc[0]) << CNOT(qi_anc
291     [0], qj_anc[0]);
292     for (auto i = 0; i < qi.size(); i++)
293     {
294         qcir << CNOT(qi[i], qj[i]) << CNOT(qj[i], qi[i]) << CNOT(qi[i], qj[i]);

```

```

294     }
295     return qcir;
296 }
297
298 QCircuit QSolver::Chebyshev(size_t n, qvec qi, qvec qi_anc, qvec qj, qvec qj_anc)
299 {
300     QCircuit qcir;
301     for (auto i = 0; i < n; i++)
302     {
303         qcir << W_circuit(qi, qi_anc, qj, qj_anc);
304     }
305     return qcir;
306 }
307 QCircuit QSolver::Chebyshev_minus(size_t n, qvec qi, qvec qi_anc, qvec qj, qvec
    qj_anc)
308 {
309     QCircuit qcir;
310     qcir << Z(qi.get()[0]) << X(qi.get()[0]) << Z(qi.get()[0]) << X(qi.get()[0]);
311     for (auto i = 0; i < n; i++)
312     {
313         qcir << W_circuit(qi, qi_anc, qj, qj_anc);
314     }
315     return qcir;
316 }
317
318 QCircuit QSolver::one_iteration_qcir(qvec qt, qvec qi, qvec qi_anc, qvec qj, qvec
    qj_anc)
319 {
320     QCircuit qcir;
321     //init |b>
322     qcir << amplitude_encode(qi, m_residual);
323     //init alpha
324     qcir << amplitude_encode(qt, m_alpha);
325     qcir << T_circuitv2(qi, qi_anc, qj, qj_anc);
326     //controlled W^n
327     QCircuit temp= Chebyshev_minus((1 << 1), qi, qi_anc, qj, qj_anc);
328     temp.setControl({ qt[0] });
329     qcir << temp;
330     for (auto i = 1; i < qt.size(); i++)
331     {
332         QCircuit qcirtemp = Chebyshev((1 << (i+1)), qi, qi_anc, qj, qj_anc);
333         qcirtemp.setControl({ qt[i] });
334         qcir << qcirtemp;
335     }
336
337     string Trname = "truncation_" + to_string((qj + qi_anc + qj_anc+qt).get().size
    ());
338     auto trun_oracle = oracle((qj + qi_anc + qj_anc+qt).get(), Trname);
339
340     QCircuit templ = Chebyshev(1, qi, qi_anc, qj, qj_anc);
341     qcir << templ;
342     qcir << T_circuitv2(qi, qi_anc, qj, qj_anc).dagger();
343     qcir << amplitude_encode(qt, m_alpha).dagger();

```

---

```
344     qcir << trun_oracle;  
345     return qcir;  
346 }  
347  
348
```