```cpp
#include <iostream>
#include "QSolver.h"
#include "SU2_CFD.hpp"

QCircuit amplitude_encode(qvec q, vector<double> data)
{
    if (data.size() > (1 << q.size()))
    {
        throw exception("error");
    }
    while(data.size() < (1 << q.size()))
    {
        data.push_back(0);
    }
    QCircuit qcir;
    double sum_0 = 0;
    double sum_1 = 0;
    size_t high_bit = (size_t)log2(data.size())-1;
    if (high_bit == 0)
    {
        if ((data[0] + data[1]) > 1e-20)
        {
            qcir << RY(q[0], 2 * acos(data[0] / sqrt(data[0] * data[0] + data[1] * data[1])));
        }
    }
    else
    {
        for (auto i = 0; i < (data.size() >> 1); i++)
        {
            sum_0 += data[i] * data[i];
            sum_1 += data[i + (data.size() >> 1)] * data[i + (data.size() >> 1)];
        }
        if (sum_0+sum_1 > 1e-20)
            qcir << RY(q[high_bit], 2 * acos(sqrt(sum_0 / (sum_0+sum_1))));
        else
        {
            throw exception("error");
        }

        if (sum_0 > 1e-20)
        {
            qvec temp({ q[high_bit] });
            vector<double> vtemp(data.begin(), data.begin() + (data.size() >> 1));
            qcir <<X(q[high_bit])<< amplitude_encode(q - temp, vtemp).control({ q[high_bit] }) << X(q[high_bit]);
        }
        if (sum_1 > 1e-20)
        {
            qvec temp({ q[high_bit] });
            vector<double> vtemp(data.begin() + (data.size() >> 1), data.end());
            qcir << amplitude_encode(q - temp, vtemp).control({ q[high_bit] });
        }
```

```cpp
    }
    return qcir;
}

QCircuit init_superposition_state(qvec q, size_t d)
{
    QCircuit qcir;


    size_t highest_bit = (size_t)log2(d-1);

    if (d == (1 << (int)log2(d)))
    {
        for (auto i = 0; i < (int)log2(d); i++)
        {
            qcir << H(q[i]);
        }
    }
    else if (d == 3)
    {
        qcir << RY(q[1], 2 * acos(sqrt(2.0 / 3))) << X(q[1]);
        qcir << H(q[0]).control({ q[1] }) << X(q[1]);
    }
    else
    {
        qcir << RY(q[highest_bit], 2 * acos(sqrt((1 << highest_bit)*1.0 / d)));
        QCircuit qcir1;
        for (auto i = 0; i < highest_bit; i++)
        {
            qcir1 << H(q[i]);
        }
        qcir << X(q[highest_bit]) << qcir1.control({ q[highest_bit] }) << X(q[highest_bit]);

        size_t d1 = d - (1 << highest_bit);
        if(d1 > 1)
        {
            QCircuit qcir2 = init_superposition_state(q, d1);
            qcir2.setControl({ q[highest_bit] });
            qcir << qcir2;
        }
    }
    return qcir;
}

QSolver::QSolver(string cfg_file) :
    m_cfg_file(cfg_file),
    m_Cheby_times(64),
    m_const_coef(100)
{
    //coefficient of Chebyshev polynomial
    m_alpha = { 1.949549963644177, -1.8488512882814025, 1.7487543978311608, -1.6496525062919445,
```

```cpp
        1.5519270299129846, -1.455943195687338, 1.3620459665535882, -1.2705563586796345,
        1.1817682156051097, -1.095945491847383, 1.0133200852492674, -0.9340902433058914,
        0.8584195544185835, -0.7864365209351393, 0.718234697381761, -0.653873364863425,
        0.5933787015467717, -0.5367453997184383, 0.48393867233793647, -0.43489658640845813,
        0.38953265692367894, -0.3477386335979911, 0.3093874129600493, -0.274336010541416,
        0.2424285316221708, -0.21349908406872772, 0.18737458295107148, -0.1638774035777065,
        0.14282784705571605, -0.12404639019677266, 0.10735569929006719, -0.09258239472103647,
        0.07955856042991463, -0.06812299861332467, 0.05812223575125098, -0.049411290903158406,
        0.04185422121218744, -0.03532446267101167, 0.02970498645435139, -0.024888292554356185,
        0.020776263132320862, -0.01727989800452769, 0.014318954104793266, -0.01182150971054103,
        0.009723472783751415, -0.007968051061520803, 0.00650519962632839, -0.005291059678290213,
        0.004287400195696609, -0.003461072169953997, 0.0027834831888454113, -0.002230098358538782,
        0.0017799719295396406, -0.0014153125440208496, 0.0011210837618425746, -0.0008846404522002947,
        0.0006954007529361845, -0.0005445525905065386, 0.00042479320707228724, -0.00033009974110103914,
        0.0002555286366486786, -0.00019704149590170955, 0.00015135492154600306, -0.00011581190277550591 };

    for (auto i = 0; i < m_alpha.size(); i++)
    {
        m_alpha[i] = sqrt(abs(m_alpha[i]));
    }
}

void QSolver::run()
{
    init();

    auto qm = initQuantumMachine(CPU_SINGLE_THREAD_WITH_ORACLE);
    Configuration config = { 10000,10000 };
    qm->setConfig(config);
    ModuleContext::setContext(qm);
    size_t qnum = ceil(log2(m_solution.size()));
    size_t qtnum = ceil(log2(m_Cheby_times));
    qvec qi(qnum);
    qvec qi_anc(1);
    qvec qj(qnum);
    qvec qj_anc(1);
    qvec qt(qtnum);
    qvec qlist = qi + qi_anc + qj + qj_anc + qt;
```

```
    QProg prog;

    prog << one_iteration_qcir(qt, qi, qi_anc, qj, qj_anc);

    string s = transformQProgToOriginIR(prog, qm);
    cout << s << endl;
    ofstream outfile;
    outfile.open("QuanutmLinearSolver.txt", ios_base::out | ios_base::trunc);
    if (!outfile.fail())
    {
        outfile << s;
        outfile.close();
    }

    qm->directlyRun(prog);
    auto target_state = qm->getQState();

    auto tmp_coef = getSquareRoot(m_solution);
    for (auto i = 0; i < m_solution.size(); i++)
    {
        m_solution[i] += tmp_coef * target_state[i].real();
    }
}

void QSolver::init()
{
    char config_file_name[MAX_STRING_SIZE];
    strcpy(config_file_name, m_cfg_file.c_str());

    CFluidDriver* driver = new CFluidDriver(config_file_name, 1, 2, false, 0, m_solution);
    driver->GetSparseMatrixAndResidual(m_sparse_matrix, m_none_zero_block, m_residual);

    delete driver;
    driver = nullptr;

    m_all_index_map = mapAllIndex(m_none_zero_block);
    m_sparse_coef = m_none_zero_block[0].size();
    for (size_t i = 1; i < m_none_zero_block.size(); i++)
    {
        if (m_sparse_coef < m_none_zero_block[i].size())
        {
            m_sparse_coef = m_none_zero_block[i].size();
        }
    }
    m_sparse_coef = m_sparse_coef * 4;
}

QCircuit QSolver::T_circuit_subspace(qvec qi,  qvec qj, qvec qj_anc)
{
    QCircuit qcir;
    size_t qtemp = (size_t)log(m_sparse_coef) + 1;
```

```cpp
    qcir << init_superposition_state(qj, m_sparse_coef);
    string OLname = "OL_" + to_string(2 * qi.get().size());
    auto OL_oracle = oracle((qi + qj).get(), OLname, m_all_index_map);
    qcir << OL_oracle;
    string OMname = "OM_" + to_string(2 * qi.get().size() + 1);
    auto OM_oracle = oracle((qi + qj + qj_anc).get(), OMname, m_sparse_matrix,
      m_none_zero_block);
    qcir << OM_oracle;
    return qcir;
}

QCircuit QSolver::T_circuit(qvec qi, qvec qi_anc, qvec qj, qvec qj_anc)
{
    QCircuit qcir;
    qcir << X(qi_anc[0]);
    QCircuit qcir1 = T_circuit_subspace(qi, qj, qj_anc);
    qcir1.setControl({ qi_anc[0] });
    qcir << qcir1;
    qcir << X(qi_anc[0]);
    qcir << X(qj_anc[0]).control({ qi_anc[0] });
    return qcir;
}

QCircuit QSolver::W_circuit(qvec qi, qvec qi_anc, qvec qj, qvec qj_anc)
{
    QCircuit qcir;
    qcir << T_circuit(qi, qi_anc, qj, qj_anc).dagger();
    for (auto i = 0; i < qj.size(); i++)
    {
        qcir << X(qj[i]);
    }
    qcir << X(qj_anc[0]);
    qcir << Z(qj_anc[0]).control(qj.get());
    for (auto i = 0; i < qj.size(); i++)
    {
        qcir << X(qj[i]);
    }
    qcir << X(qj_anc[0]);
    qcir << Z(qj_anc[0]) << X(qj_anc[0]) << Z(qj_anc[0]) << X(qj_anc[0]);

    qcir << T_circuit(qi, qi_anc, qj, qj_anc);

    qcir << CNOT(qi_anc[0], qj_anc[0])<< CNOT(qj_anc[0], qi_anc[0])<< CNOT(qi_anc
      [0], qj_anc[0]);
    for (auto i = 0; i < qi.size(); i++)
    {
        qcir << CNOT(qi[i], qj[i]) << CNOT(qj[i], qi[i]) << CNOT(qi[i], qj[i]);
    }
    return qcir;
}

QCircuit QSolver:: Chebyshev(size_t n, qvec qi, qvec qi_anc, qvec qj, qvec qj_anc)
{
```

```
    QCircuit qcir;
    for (auto i = 0; i < n; i++)
    {
        qcir << W_circuit(qi, qi_anc, qj, qj_anc);
    }
    return qcir;
}

QCircuit QSolver::Chebyshev_minus(size_t n, qvec qi, qvec qi_anc, qvec qj, qvec qj_anc)
{
    QCircuit qcir;
    qcir << Z(qi.get()[0]) << X(qi.get()[0]) << Z(qi.get()[0]) << X(qi.get()[0]);
    for (auto i = 0; i < n; i++)
    {
        qcir << W_circuit(qi, qi_anc, qj, qj_anc);
    }
    return qcir;
}

QCircuit QSolver::one_iteration_qcir(qvec qt, qvec qi, qvec qi_anc, qvec qj, qvec qj_anc)
{
    QCircuit qcir;
    //init |b>
    qcir << amplitude_encode(qi, m_residual);
    //init alpha
    qcir << amplitude_encode(qt, m_alpha);
    qcir << T_circuit(qi, qi_anc, qj, qj_anc);
    //controlled W^n
    QCircuit temp= Chebyshev_minus((1 << 1), qi, qi_anc, qj, qj_anc);
    temp.setControl({ qt[0] });
    qcir << temp;
    for (auto i = 1; i < qt.size(); i++)
    {
        QCircuit qcirtemp = Chebyshev((1 << (i+1)), qi, qi_anc, qj, qj_anc);
        qcirtemp.setControl({ qt[i] });
        qcir << qcirtemp;
    }

    string Trname = "truncation_" + to_string((qj + qi_anc + qj_anc+qt).get().size());
    auto trun_oracle = oracle((qj + qi_anc + qj_anc+qt).get(), Trname);

    QCircuit temp1 = Chebyshev(1, qi, qi_anc, qj, qj_anc);
    qcir << temp1;
    qcir << T_circuit(qi, qi_anc, qj, qj_anc).dagger();
    qcir << amplitude_encode(qt, m_alpha).dagger();
    qcir << trun_oracle;
    return qcir;
}

vector<vector<size_t>> QSolver::mapAllIndex(const vector<vector<size_t>>& data)
```

```
{
    vector<vector<size_t>> post_data;
    for (auto i = 0; i < 4*data.size(); i++)
    {
        auto row = i / 4;
        vector<size_t> vtemp;
        for (auto j = 0; j < 4*data[row].size(); j++)
        {
            vtemp.push_back(4 * data[row][j / 4] + j % 4);
        }
        post_data.push_back(vtemp);
    }

    vector<vector<size_t>> new_data = post_data;

    for (size_t i = 0; i < post_data.size(); i++)
    {
        for (size_t j = 0; j < post_data.size(); j++)
        {
            if (find(post_data[i].begin(), post_data[i].end(), j) == post_data[i].end())
            {
                new_data[i].push_back(j);
            }
        }
    }

    return new_data;
}

double QSolver::getSquareRoot(const vector<double>& data)
{
    double sum = 0;

    for (size_t i = 0; i < data.size(); i++)
    {
        sum += data[i] * data[i];
    }

    return sqrt(sum);
}

```