## **一. 测试目的**

测试对象SCIP在C/C++环境下的性能。

因版权问题，无法对Gurobi 等商用软件进行测试。

## **二. 测试平台**

Processor Intel(R) Xeon(R) w9-3495X 1.90 GHz

Installed RAM 256 GB (255 GB usable)

System type 64-bit operating system, x64-based processor

Edition Windows 11 Pro for Workstations

Version 22H2

Installed on ‎10/‎4/‎2024

OS build 22621.4317

Experience Windows Feature Experience Pack 1000.22700.1041.0

## **三. 软件平台**

|  |  |
| --- | --- |
| 编译器 | VS code |
| Python 版本 | 3.8.x |
| OS | Windows |
| CPU 核心数 | 56核心 112线程 |
| 测试时间 | 29 Oct 2024 |

## **四. 代码**

#include <iostream>

#include <vector>

#include <string>

#include <scip/scip.h>

#include <scip/scipdefplugins.h>

#include <random>

#include <fstream>

// 参数初始化

const int clear\_interval = 30; // 分钟

const int clear\_period = 24 \* 600; // 优化周期，分钟

const int ctrl\_interval = 1; // 控制变量步长，分钟

const double charge\_eff = 0.91;

const double discharge\_eff = 0.95;

const double nominal\_power = 0.8;

const double SOC\_ub = 1;

const double SOC\_lb = 0;

const double SOC0 = 0.5;

const double Ckwh = 1; // 电池容量 (kWh)

int main()

{

    // 初始化 SCIP

    SCIP\* scip = nullptr;

    SCIP\_CALL(SCIPcreate(&scip));

    SCIP\_CALL(SCIPincludeDefaultPlugins(scip));

    SCIP\_CALL(SCIPcreateProb(scip, "Electricity Optimization", nullptr, nullptr, nullptr, nullptr, nullptr, nullptr, nullptr));

    SCIP\_CALL(SCIPsetObjsense(scip, SCIP\_OBJSENSE\_MINIMIZE));

    // 生成随机数据（与Python代码一致）

    std::vector<double> user\_loads, user\_powers, elec\_price;

    std::default\_random\_engine generator(1234);

    std::uniform\_real\_distribution<double> distribution(0.0, 1.0);

    int n\_clearance = clear\_period / clear\_interval;

    int n\_ctrl = clear\_period / ctrl\_interval;

    // 用户负荷和功率

    for (int i = 0; i < n\_clearance; ++i) {

        double load = round(distribution(generator) \* 100) / 100.0;

        double power = round(distribution(generator) \* 100) / 100.0;

        for (int j = 0; j < clear\_interval / ctrl\_interval; ++j) {

            user\_loads.push\_back(load);

            user\_powers.push\_back(power);

        }

    }

    // 生成净负荷数据

    std::vector<double> netload;

    for (int i = 0; i < n\_ctrl; ++i) {

        double load = user\_loads[i];

        double power = user\_powers[i];

        double net = load - power;

        netload.push\_back(net);

    }

    // 打印净负荷值

    std::cout << "净负荷：" << std::endl;

    for (size\_t i = 0; i < 100; ++i) {

        std::cout << "netload[" << i << "] = " << netload[i] << std::endl;

    }

    // 电价

    for (int i = 0; i < n\_clearance; i += 12) {

        double price = round((0.5 + distribution(generator) \* 0.4 - 0.2) \* 100) / 100.0;

        for (int j = 0; j < 12 \* (clear\_interval / ctrl\_interval); ++j) {

            elec\_price.push\_back(price);

        }

    }

    // 初始化决策变量

    std::vector<SCIP\_VAR\*> x\_vars, y\_vars, soc\_vars, z\_vars;

    for (int i = 0; i < n\_ctrl; ++i) {

        SCIP\_VAR\* x;

        SCIP\_VAR\* y;

        SCIP\_VAR\* soc;

        SCIP\_VAR\* z;

        // 创建变量

        SCIP\_CALL(SCIPcreateVarBasic(scip, &x, ("x\_" + std::to\_string(i)).c\_str(), 0.0, nominal\_power, 0.0, SCIP\_VARTYPE\_CONTINUOUS));

        SCIP\_CALL(SCIPcreateVarBasic(scip, &y, ("y\_" + std::to\_string(i)).c\_str(), -nominal\_power, 0.0, 0.0, SCIP\_VARTYPE\_CONTINUOUS));

        SCIP\_CALL(SCIPcreateVarBasic(scip, &soc, ("soc\_" + std::to\_string(i)).c\_str(), SOC\_lb, SOC\_ub, 0.0, SCIP\_VARTYPE\_CONTINUOUS));

        SCIP\_CALL(SCIPcreateVarBasic(scip, &z, ("z\_" + std::to\_string(i)).c\_str(), 0.0, 1.0, 0.0, SCIP\_VARTYPE\_BINARY));

        // 添加变量到模型

        SCIP\_CALL(SCIPaddVar(scip, x));

        SCIP\_CALL(SCIPaddVar(scip, y));

        SCIP\_CALL(SCIPaddVar(scip, soc));

        SCIP\_CALL(SCIPaddVar(scip, z));

        x\_vars.push\_back(x);

        y\_vars.push\_back(y);

        soc\_vars.push\_back(soc);

        z\_vars.push\_back(z);

    }

    // 设置目标函数（与Python代码一致）

    for (int i = 0; i < n\_ctrl; ++i) {

        double price = elec\_price[i];

        SCIP\_CALL(SCIPchgVarObj(scip, x\_vars[i], price));

        SCIP\_CALL(SCIPchgVarObj(scip, y\_vars[i], price));

    }

    // 初始 SOC 约束

    SCIP\_CONS\* cons;

    SCIP\_CALL(SCIPcreateConsBasicLinear(scip, &cons, "soc\_initial", 1, &soc\_vars[0], (SCIP\_Real[]){1.0}, SOC0, SOC0));

    SCIP\_CALL(SCIPaddCons(scip, cons));

    SCIP\_CALL(SCIPreleaseCons(scip, &cons));

    // 更新 SOC 和充放电约束

    for (int i = 0; i < n\_ctrl - 1; ++i) {

        SCIP\_VAR\* vars[] = {soc\_vars[i], soc\_vars[i + 1], x\_vars[i], y\_vars[i]};

        SCIP\_Real coeffs[] = {1.0, -1.0, charge\_eff \* ctrl\_interval / 60 / Ckwh, ctrl\_interval / discharge\_eff / 60 / Ckwh};

        SCIP\_CALL(SCIPcreateConsBasicLinear(scip, &cons, ("soc\_update\_" + std::to\_string(i)).c\_str(), 4, vars, coeffs, 0.0, 0.0));

        SCIP\_CALL(SCIPaddCons(scip, cons));

        SCIP\_CALL(SCIPreleaseCons(scip, &cons));

        // 平滑充放电约束

        SCIP\_CALL(SCIPcreateConsBasicLinear(scip, &cons, ("x\_smooth\_" + std::to\_string(i)).c\_str(), 2, (SCIP\_VAR\*[]){x\_vars[i + 1], x\_vars[i]}, (SCIP\_Real[]){1.0, -1.0}, -0.01, 0.01));

        SCIP\_CALL(SCIPaddCons(scip, cons));

        SCIP\_CALL(SCIPreleaseCons(scip, &cons));

        SCIP\_CALL(SCIPcreateConsBasicLinear(scip, &cons, ("y\_smooth\_" + std::to\_string(i)).c\_str(), 2, (SCIP\_VAR\*[]){y\_vars[i + 1], y\_vars[i]}, (SCIP\_Real[]){1.0, -1.0}, -0.01, 0.01));

        SCIP\_CALL(SCIPaddCons(scip, cons));

        SCIP\_CALL(SCIPreleaseCons(scip, &cons));

    }

    // 充放电互斥约束

    for (int i = 0; i < n\_ctrl; ++i) {

        SCIP\_VAR\* charge\_vars[] = {x\_vars[i], z\_vars[i]};

        SCIP\_Real charge\_coeffs[] = {1.0, -nominal\_power};

        SCIP\_CALL(SCIPcreateConsBasicLinear(scip, &cons, ("charge\_exclusivity\_" + std::to\_string(i)).c\_str(), 2, charge\_vars, charge\_coeffs, -SCIPinfinity(scip), 0.0));

        SCIP\_CALL(SCIPaddCons(scip, cons));

        SCIP\_CALL(SCIPreleaseCons(scip, &cons));

        SCIP\_VAR\* discharge\_vars[] = {y\_vars[i], z\_vars[i]};

        SCIP\_Real discharge\_coeffs[] = {1.0, nominal\_power};

        SCIP\_CALL(SCIPcreateConsBasicLinear(scip, &cons, ("discharge\_exclusivity\_" + std::to\_string(i)).c\_str(), 2, discharge\_vars, discharge\_coeffs, -SCIPinfinity(scip), 0.0));

        SCIP\_CALL(SCIPaddCons(scip, cons));

        SCIP\_CALL(SCIPreleaseCons(scip, &cons));

    }

    // 求解模型

    SCIP\_CALL(SCIPsolve(scip));

    SCIP\_SOL\* solution = SCIPgetBestSol(scip);

    if (solution != nullptr) {

        SCIP\_Real objective\_value = SCIPgetSolOrigObj(scip, solution);

        std::cout << "Optimized objective function value: " << objective\_value << std::endl;

        std::ofstream result\_file("optimization\_results.csv");

        result\_file << "Time,User Load,User Power,Charge,Discharge,SOC\n";

        for (int i = 0; i < n\_ctrl; ++i) {

            double x\_val = SCIPgetSolVal(scip, solution, x\_vars[i]);

            double y\_val = SCIPgetSolVal(scip, solution, y\_vars[i]);

            double soc\_val = SCIPgetSolVal(scip, solution, soc\_vars[i]);

            result\_file << i << "," << user\_loads[i] << "," << user\_powers[i] << "," << x\_val << "," << y\_val << "," << soc\_val << "\n";

        }

        result\_file.close();

        std::cout << "Results saved to 'optimization\_results.csv'\n";

    } else {

        std::cout << "No feasible solution found.\n";

    }

    // 释放所有决策变量

    for (SCIP\_VAR\* var : x\_vars) {

        SCIP\_CALL(SCIPreleaseVar(scip, &var));

    }

    for (SCIP\_VAR\* var : y\_vars) {

        SCIP\_CALL(SCIPreleaseVar(scip, &var));

    }

    for (SCIP\_VAR\* var : soc\_vars) {

        SCIP\_CALL(SCIPreleaseVar(scip, &var));

    }

    for (SCIP\_VAR\* var : z\_vars) {

        SCIP\_CALL(SCIPreleaseVar(scip, &var));

    }

    // 释放 SCIP 问题

    SCIP\_CALL(SCIPfree(&scip));

    return 0;

}

## **五. 主要输出：**

SCIP Status : problem is solved [optimal solution found]

Solving Time (sec) : 84.00

Solving Nodes : 1

Primal Bound : -1.96688999999999e+01 (1 solutions)

Dual Bound : -1.96688999999999e+01

Gap : 0.00 %

Optimized objective function value: -19.6689

## **六. 结论**

1. 开源求解器SCIP可以在C/C++编译环境下运行，建模精度、计算速度和求解准确度与Python一致。

2. 在不同的编译环境下，主要时间差异在于优化问题的建模，实际求解均调用基于求解器底层语言与计算机硬件交互，求解速度不受编译环境限制。以下是详细信息：

|  |  |  |  |
| --- | --- | --- | --- |
| **求解器** | **编程语言** | **常用接口** | **与计算机的互动** |
| **SCIP** | C（核心），C++，Python | C、C++、Python | 直接与CPU和内存交互，通过优化数值库（如Soplex） |
| **CBC** | C++ | C++、Python（PuLP） | 直接内存和CPU操作，通过COIN-OR库交互 |
| **CPLEX** | C、C++ | Python、Java、C#、MATLAB | 优化的并行计算和多核利用，通过高效算法直接调用CPU和内存资源 |
| **Gurobi** | C、C++ | Python、Java、MATLAB | 高度并行化，优化的内存和CPU利用，支持多线程并行计算 |

3. 从时间复杂度考虑，Python和C/C++对于优化求解问题处于同一量级；从建模效率和代码工程考虑，Python更为友好。建议项目以Python编译环境作为后端开发和求解器接口，以C/C++作为底层调用工具。