```cpp
//
//  SOCOPF.cpp
//  Gravity
//
//  Created by Guanglei Wang on 6/9/17.
//
//
#include <stdio.h>
#include <iostream>
#include <string>
#include <stdio.h>
#include <cstring>
#include <fstream>
#include "../PowerNet.h"
#include <gravity/solver.h>
#include <stdio.h>
#include <stdlib.h>
#include <optionParser.hpp>

using namespace std;
using namespace gravity;

int main (int argc, char * argv[])
{
    int output = 0;
    bool relax = false, use_cplex = false, use_gurobi = false;
    double tol = 1e-6;
    double solver_time_end, total_time_end, solve_time, total_time;
    string mehrotra = "no", log_level="0";
    string fname = "../data_sets/Power/nesta_case5_pjm.m";

    string path = argv[0];
    string solver_str="ipopt";

    /** Create a OptionParser with options */
    op::OptionParser opt;
    opt.add_option("h", "help", "shows option help"); // no default value means
            boolean options, which default value is false
    opt.add_option("f", "file", "Input file name", fname);
    opt.add_option("l", "log", "Log level (def. 0)", log_level );
    opt.add_option("s", "solver", "Solvers: ipopt/cplex/gurobi, default = ipopt",
            solver_str);

    /** Parse the options and verify that all went well. If not, errors and help
            will be shown */
    bool correct_parsing = opt.parse_options(argc, argv);

    if(!correct_parsing){
        return EXIT_FAILURE;
    }

    output = op::str2int(opt["l"]);

    fname = opt["f"];
    bool has_help = op::str2bool(opt["h"]);
    if(has_help) {
```

```
    opt.show_help();
    exit(0);
}
solver_str = opt["s"];
if (solver_str.compare("gurobi")==0) {
    use_gurobi = true;
}
else if(solver_str.compare("cplex")==0) {
    use_cplex = true;
}
double total_time_start = get_wall_time();
PowerNet* grid = new PowerNet();
grid->readgrid(fname.c_str());

/* Grid Parameters */
auto bus_pairs = grid->get_bus_pairs();
auto nb_bus_pairs = grid->get_nb_active_bus_pairs();
auto nb_gen = grid->get_nb_active_gens();
auto nb_lines = grid->get_nb_active_arcs();
auto nb_buses = grid->get_nb_active_nodes();
DebugOn("nb gens = " << nb_gen << endl);
DebugOn("nb lines = " << nb_lines << endl);
DebugOn("nb buses = " << nb_buses << endl);
DebugOn("nb bus_pairs = " << nb_bus_pairs << endl);

/** Build model */
Model SOCP("SOCP Model");

/** Variables */
/* power generation variables */
var<Real> Pg("Pg", grid->pg_min, grid->pg_max);
var<Real> Qg ("Qg", grid->qg_min, grid->qg_max);
SOCP.add_var(Pg.in(grid->gens));
SOCP.add_var(Qg.in(grid->gens));


/* power flow variables */
var<Real> Pf_from("Pf_from", grid->S_max);
var<Real> Qf_from("Qf_from", grid->S_max);
var<Real> Pf_to("Pf_to", grid->S_max);
var<Real> Qf_to("Qf_to", grid->S_max);
SOCP.add_var(Pf_from.in(grid->arcs));
SOCP.add_var(Qf_from.in(grid->arcs));
SOCP.add_var(Pf_to.in(grid->arcs));
SOCP.add_var(Qf_to.in(grid->arcs));

/* Real part of Wij = ViVj */
var<Real>  R_Wij("R_Wij", grid->wr_min, grid->wr_max);
/* Imaginary part of Wij = ViVj */
var<Real>  Im_Wij("Im_Wij", grid->wi_min, grid->wi_max);
/* Magnitude of Wii = Vi^2 */
var<Real>  Wii("Wii", grid->w_min, grid->w_max);
SOCP.add_var(Wii.in(grid->nodes));
SOCP.add_var(R_Wij.in(bus_pairs));
SOCP.add_var(Im_Wij.in(bus_pairs));
```

```
/* Initialize variables */
R_Wij.initialize_all(1.0);
Wii.initialize_all(1.001);

/**  Objective */
auto obj = product(grid->c1, Pg) + product(grid->c2, power(Pg,2)) + sum(grid-
        >c0);
SOCP.min(obj.in(grid->gens));

/** Constraints */
/* Second-order cone constraints */
Constraint SOC("SOC");
SOC = power(R_Wij, 2) + power(Im_Wij, 2) - Wii.from()*Wii.to();
SOCP.add_constraint(SOC.in(bus_pairs) <= 0);

/* Flow conservation */
Constraint KCL_P("KCL_P");
KCL_P  = sum(Pf_from.out_arcs()) + sum(Pf_to.in_arcs()) + grid->pl - sum(Pg.
        in_gens()) + grid->gs*Wii;
SOCP.add_constraint(KCL_P.in(grid->nodes) == 0);

Constraint KCL_Q("KCL_Q");
KCL_Q  = sum(Qf_from.out_arcs()) + sum(Qf_to.in_arcs()) + grid->ql - sum(Qg.
        in_gens()) - grid->bs*Wii;
SOCP.add_constraint(KCL_Q.in(grid->nodes) == 0);

/* AC Power Flow */
Constraint Flow_P_From("Flow_P_From");
Flow_P_From = Pf_from - (grid->g_ff*Wii.from() + grid->g_ft*R_Wij.in_pairs()
        + grid->b_ft*Im_Wij.in_pairs());
SOCP.add_constraint(Flow_P_From.in(grid->arcs) == 0);

Constraint Flow_P_To("Flow_P_To");
Flow_P_To = Pf_to - (grid->g_tt*Wii.to() + grid->g_tf*R_Wij.in_pairs() - grid
        ->b_tf*Im_Wij.in_pairs());
SOCP.add_constraint(Flow_P_To.in(grid->arcs) == 0);

Constraint Flow_Q_From("Flow_Q_From");
Flow_Q_From = Qf_from - (grid->g_ft*Im_Wij.in_pairs() - grid->b_ff*Wii.from()
        - grid->b_ft*R_Wij.in_pairs());
SOCP.add_constraint(Flow_Q_From.in(grid->arcs) == 0);

Constraint Flow_Q_To("Flow_Q_To");
Flow_Q_To = Qf_to + (grid->b_tt*Wii.to() + grid->b_tf*R_Wij.in_pairs() + grid
        ->g_tf*Im_Wij.in_pairs());
SOCP.add_constraint(Flow_Q_To.in(grid->arcs) == 0);

/* Phase Angle Bounds constraints */
Constraint PAD_UB("PAD_UB");
PAD_UB = Im_Wij;
PAD_UB <= grid->tan_th_max*R_Wij;
SOCP.add_constraint(PAD_UB.in(bus_pairs));

Constraint PAD_LB("PAD_LB");
PAD_LB =  Im_Wij;
PAD_LB >= grid->tan_th_min*R_Wij;
```

```
SOCP.add_constraint(PAD_LB.in(bus_pairs));

/* Thermal Limit Constraints */
Constraint Thermal_Limit_from("Thermal_Limit_from");
Thermal_Limit_from = power(Pf_from, 2) + power(Qf_from, 2);
Thermal_Limit_from <= power(grid->S_max,2);
SOCP.add_constraint(Thermal_Limit_from.in(grid->arcs));


Constraint Thermal_Limit_to("Thermal_Limit_to");
Thermal_Limit_to = power(Pf_to, 2) + power(Qf_to, 2);
Thermal_Limit_to <= power(grid->S_max,2);
SOCP.add_constraint(Thermal_Limit_to.in(grid->arcs));

/* Lifted Nonlinear Cuts */
Constraint LNC1("LNC1");
LNC1 += (grid->v_min.from()+grid->v_max.from())*(grid->v_min.to()+grid->v_max
        .to())*(grid->sphi*Im_Wij + grid->cphi*R_Wij);
LNC1 -= grid->v_max.to()*grid->cos_d*(grid->v_min.to()+grid->v_max.to())*Wii.
        from();
LNC1 -= grid->v_max.from()*grid->cos_d*(grid->v_min.from()+grid->v_max.from()
        )*Wii.to();
LNC1 -= grid->v_max.from()*grid->v_max.to()*grid->cos_d*(grid->v_min.from()*
        grid->v_min.to() - grid->v_max.from()*grid->v_max.to());
SOCP.add_constraint(LNC1.in(bus_pairs) >= 0);

Constraint LNC2("LNC2");
LNC2 += (grid->v_min.from()+grid->v_max.from())*(grid->v_min.to()+grid->v_max
        .to())*(grid->sphi*Im_Wij + grid->cphi*R_Wij);
LNC2 -= grid->v_min.to()*grid->cos_d*(grid->v_min.to()+grid->v_max.to())*Wii.
        from();
LNC2 -= grid->v_min.from()*grid->cos_d*(grid->v_min.from()+grid->v_max.from()
        )*Wii.to();
LNC2 += grid->v_min.from()*grid->v_min.to()*grid->cos_d*(grid->v_min.from()*
        grid->v_min.to() - grid->v_max.from()*grid->v_max.to());
SOCP.add_constraint(LNC2.in(bus_pairs) >= 0);


/* Solver selection */
/* TODO: declare only one solver and one set of time measurment functions for
        all solvers. */
if (use_cplex) {
    solver SCOPF_CPX(SOCP, cplex);
    auto solver_time_start = get_wall_time();
    SCOPF_CPX.run(output = 0, relax = false, tol = 1e-6);
    solver_time_end = get_wall_time();
    total_time_end = get_wall_time();
    solve_time = solver_time_end - solver_time_start;
    total_time = total_time_end - total_time_start;
}
else if (use_gurobi) {
    solver SCOPF_GRB(SOCP, gurobi);
    auto solver_time_start = get_wall_time();
    SCOPF_GRB.run(output, relax = false, tol = 1e-6);
    solver_time_end = get_wall_time();
    total_time_end = get_wall_time();
```

```
        solve_time = solver_time_end - solver_time_start;
        total_time = total_time_end - total_time_start;
    }
    else {
        solver SCOPF(SOCP,ipopt);
        auto solver_time_start = get_wall_time();
        SCOPF.run(output, relax = false, tol = 1e-6, "ma27", mehrotra = "no");
        solver_time_end = get_wall_time();
        total_time_end = get_wall_time();
        solve_time = solver_time_end - solver_time_start;
        total_time = total_time_end - total_time_start;
    }
    /** Uncomment next line to print expanded model */
    /* SOCP.print_expanded(); */
    string out = "DATA_OPF, " + grid->_name + ", " + to_string(nb_buses) + ", " +
             to_string(nb_lines) +", " + to_string(SOCP._obj_val) + ", " +
             to_string(-numeric_limits<double>::infinity()) + ", " + to_string
             (solve_time) + ", LocalOptimal, " + to_string(total_time);
    DebugOn(out <<endl);
    return 0;
}
```