Help

```
#include <iostream>
#include <fstream>
#include <cmath>
#include <cstdlib>
#include <cstring>
#include "cxxopts.hpp"
#include <boost/date_time/posix_time/posix_time.hpp>
#include <mpi.h>
#include "
href../../../../common/math/jlparser/include/jlparser/parser_h_src.pdfjlparser/p
#include "
href../../../../common/math/mcam/src/MonteCarlo_h_src.pdfMonteCarlo.hpp"
#include "
href../../../../common/math/mcam/src/optim_h_src.pdfoptim.hpp"

#include "pnl/pnl_random.h"

#ifndef _OPENMP
double omp_get_wtime()
{
    boost::posix_time::ptime ancestor(boost::gregorian::date(2016,3,1));
    boost::posix_time::ptime t = boost::posix_time::microsec_clock::local_time()
    return double((t - ancestor).total_milliseconds()) / 1000.;
}
#endif

using namespace std;


int main(int argc, char **argv)
{
    mcam::Option *opt = NULL;
    mcam::Model *mod = NULL;
    mcam::Martingale *mart = NULL;
    mcam::Optim *optim = NULL;
    bool useBlocks;
    bool verbose, dpp, dual, rnd_seed, prune;
    string regressionTypeString;
    string optimTypeString;
```

```
string infile;
int loops, rank;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

cxxopts::Options options("mc-pricer", "Hedging tool");

options.add_options()
("help", "Print help message")
("block", "Use increasing size blocks.", cxxopts::value<bool>())
("optim", "Type of optimization algorithm: sg, saa, saa-avg, sg-averaging, s
("dual", "Compute the dual price.", cxxopts::value<bool>())
("dpp", "Solve the DPP using a regression expansion without control variate.
("rnd-seed", "Use a random seed.", cxxopts::value<bool>())
("prune", "Prune the chaos representation.", cxxopts::value<bool>())
("infile", "Path to the input file.", cxxopts::value<std::string>())
("regression-type", "Type of regression. Can be pol, chaos, reduced_chaos.",
("loops", "Number of algorithm runs. Default=1.", cxxopts::value<int>()->def
("verbose", "Print the parameters.", cxxopts::value<bool>())
;

auto vm = options.parse(argc, argv);

optimTypeString = vm["optim"].as<std::string>();
useBlocks = vm["block"].as<bool>();
dual = vm["dual"].as<bool>();
dpp = vm["dpp"].as<bool>();
prune = vm["prune"].as<bool>();
infile = vm["infile"].as<std::string>();
rnd_seed = vm["rnd-seed"].as<bool>();
verbose = vm["verbose"].as<bool>();
loops = vm["loops"].as<int>();
regressionTypeString = vm["regression-type"].as<std::string>();

if (vm.count("help") || !vm.count("infile"))
{
    std::cout << options.help() << "\ n";
    return 1;
}
if (dpp && !vm.count("regression-type"))
```

```
{
    std::cout << "Make sure to specify the type of regressors." << std::endl
    std::cout << options.help() << "\ n";
    return 1;
}
if (dpp && regressionTypeString == "pol")
{
    std::cout << "Polynomial regression is not supported in parallel mode."
    std::cout << options.help() << "\ n";
    return 1;
}


if (rank == 0)
{
    if (dual) std::cout << "--dual" << std::endl;
    if (dpp)
    {
        std::cout << "--dpp" << std::endl;
        std::cout << "--regression-type=" << regressionTypeString << std::en
    }
    std::cout << "--infile=" << infile << std::endl;
    std::cout << "--optim=" << optimTypeString << std::endl;
    if (loops > 1) std::cout << "--loops=" << loops << "\ n";
    if (useBlocks) std::cout << "--block" << std::endl;
    if (rnd_seed) std::cout << "--rnd-seed" << std::endl;
    int n_procs;
    MPI_Comm_size(MPI_COMM_WORLD, &n_procs);
    std::cout << "Number of processes: " << n_procs << std::endl;
}

// Create the instances of the problem
mcam::PnlRng_Workspace rng;
if (! rnd_seed) rng.set_seed(0);
Parser map(infile.c_str());

if ((mod = mcam::instantiate_model(map)) == NULL) abort();
if ((opt = mcam::instantiate_option(map)) == NULL) abort();
if (dual)
{
    if ((mart = mcam::instantiate_martingale(mod, map, prune)) == NULL) abor
```

```
    if ((optim = mcam::instantiate_optim(mod, opt, mart, map)) == NULL) abor
}

mcam::MonteCarlo mc(mod, opt, mart, map, regressionTypeString);

// Print the input data
if (verbose)
{
    mod->print();
    opt->print();
    if (mart) mart->print();
    if (optim) optim->print();
    mc.print();
}

double prix, var;
double prix_moyen = 0, varinline_moyen = 0, var_true = 0;
for (int l = 0; l < loops; l++)
{
    if (dual)
    {
        PnlVect *alpha = pnl_vect_create_from_zero(mart->nbFreedom);
        double start = MPI_Wtime();
        if (optimTypeString == "saa" || optimTypeString == "saa-explore")
        {
            mcam::StepMethod *stepMethod = NULL;
            if (optimTypeString == "saa")
                stepMethod = new mcam::LineSearchStep();
            else if (optimTypeString == "saa-explore")
                stepMethod = new mcam::ExploreStep();
            optim->saa(rng, alpha, prix, var, stepMethod, useBlocks);
            delete stepMethod;
        }
        else
        {
            std::cout << "Value " << optimTypeString << " is unknown." << st
            std::cout << options.help() << "\ n";
            return 1;
        }
        double end = MPI_Wtime();
        if (rank == 0)
```

```
        {
            std::cout << "Price (SAA): " << prix << std::endl;
            std::cout << "Standard deviation (SAA): " << std::sqrt(var) << s
            cout << "Time for the optimization step: " << end - start << end
            double grad_cost;
            optim->EGradCost(rng, alpha, grad_cost, prix, var, false);
            // cout << "grad_cost: " << grad_cost << std::endl;
            cout << "Dual price: " << prix << std::endl;
            cout << "Dual standard deviation: " << std::sqrt(var) << std::en
        }
        pnl_vect_free(&alpha);
    }
    if (dpp)
    {
        double start = MPI_Wtime();
        mc.backward_price_dpp(prix, var, NULL, rng);
        double end = MPI_Wtime();
        if (rank == 0)
        {
            cout << "DPP price, variance: " << prix << "\ t" << var << endl;
            cout << "Time for the DPP resolution step: " << end - start << e
        }
    }
    if (rank ==0)
    {
        prix_moyen += prix;
        varinline_moyen += var;
        var_true += prix * prix;
    }
}

if ((rank == 0) && (loops > 1))
{
    prix_moyen /= loops;
    varinline_moyen /= loops;
    var_true = var_true / loops - prix_moyen * prix_moyen;
    cout << "Mean price over " << loops << " runs: " << prix_moyen << "\ n";
    cout << "Var price over " << loops << " runs: " << var_true << "\ n";
}
```

```
    // Clean memory
    delete mod;
    delete opt;
    if (mart) delete mart;
    if (optim) delete optim;

    MPI_Finalize();
    exit(0);
}
```