Help

```
#include "
href../../../../common/math/mcam/src/optim_h_src.pdfoptim.hpp"
#include <cmath>
#include <string>
#include <iostream>
#ifdef HAVE_MPI
#include "pnl/pnl_mpi.h"
#endif
#ifdef _OPENMP
#include <omp.h>
#endif

/**
 * Decide whether the new proposal is admissible. If not, divide the step by 2.
 * does not move enough anymore, send the stop signal.
 *
 * @param cost the new cost
 * @param step [in,out] the value of the step
 * @param best_cost the best cost sofar
 *
 * @return mcam::StepMethod::Accept, mcam::StepMethod::Reject, mcam::StepMethod:
 */
int  mcam::LineSearchStep::Step(double cost, double &step, double &best_cost)
{
    double norm_eps = 1E-3;
    if (cost < best_cost)
    {
        if ( (cost < best_cost) && ((best_cost - cost) / best_cost < norm_eps))
        {
            best_cost = cost;
            return Stop;
        }
        best_cost = cost;
        return Accept;
    }
    else
    {
        // We have rejected the last four moves
        if (step < 0.25) return Stop;
```

```
        step /= 2.;
        std::cout << "Reject move" << std::endl;
        return Reject;
    }
}

/**
 * Always accept any new proposal. It enables to explore the space while keeping
 * best cost, somehow similar to a subgradient approach.
 *
 * @param cost the new cost
 * @param step [in,out] the value of the step
 * @param best_cost the best cost sofar
 *
 * @return mcam::StepMethod::Accept, mcam::StepMethod::Reject, mcam::StepMethod:
 */
int mcam::ExploreStep::Step(double cost, double &step, double &best_cost)
{
    if (cost < best_cost)
        best_cost = cost;
    step = 1.;
    return Accept;
}


mcam::Optim *mcam::instantiate_optim(mcam::Model *mod, mcam::Option *opt, mcam::
{
    return  new mcam::MaxCost(mod, opt, mart, P);
}

mcam::Optim::Optim()
    : mod(NULL), opt(NULL), mart(NULL),
      iterationsSA(0), iterationsMC(0), block_SA(0),
      gamma(0) { }

mcam::Optim::Optim(mcam::Model *mod, mcam::Option *opt, mcam::Martingale *mart,
    : mod(mod), opt(opt), mart(mart)
{
    useMPI = false;
#ifdef HAVE_MPI
    int tmp;
```

```
    MPI_Initialized(&tmp);
    useMPI = (tmp == 1);
#endif
    P.extract("gamma step", gamma, true);
    P.extract("SA iterations", iterationsSA, true);
    P.extract("MC iterations", iterationsMC);
    if (P.extract("SAA iterations", iterationsSAA, true) == false) iterationsSAA
    if (P.extract("SA block size", block_SA, true) == false) block_SA = 1;
}

void mcam::Optim::print() const
{

    std::cout << std::endl;
    std::cout << "******************************" << std::endl;
    std::cout << "**** Optim Characteristics ****" << std::endl;
    std::cout << " gamma step " << gamma << std::endl;
    std::cout << " SA iterations " << iterationsSA << std::endl;
    std::cout << " MC iterations " << iterationsMC << std::endl;
    std::cout << " SA block size " << block_SA << std::endl;
}

/**
 * Compute the price of the European option
 *
 * @param rng
 *
 */
double mcam::Optim::computeEuropenPrice(mcam::PnlRng_Workspace &rng)
{
    double price = 0.;
    int nSamples = 5000;
    PnlMat *DeltaB = pnl_mat_new();
    for (int i = 0; i < nSamples; i++)
    {
        pnl_mat_rng_normal(DeltaB, mart->subdates, mart->size, rng());
        mod->path(DeltaB);
        price += opt->payoff(mod->getPath(), mod->subdates);
    }
    pnl_mat_free(&DeltaB);
    return price * mod->discount(mod->subdates) / double(nSamples);
```

```
}

/**
 * Compute the probability that an American option is in the money at one of the
 *
 * @param rng a random number generator
 *
 * @return a probability
 */
double mcam::Optim::computeMoneyness(PnlRng_Workspace &rng)
{
    int count = 0;
    int nSamples = 500;
    PnlMat *DeltaB = pnl_mat_new();
    for (int i = 0; i < nSamples; i++)
    {
        pnl_mat_rng_normal(DeltaB, mart->subdates, mart->size, rng());
        mod->path(DeltaB); // sample the model
        for (int t_sub = 0; t_sub <= mod->subdates; t_sub += mod->subticks)
        {
            double Zt = opt->payoff(mod->getPath(), t_sub);
            if (Zt > 0)
            {
                count ++;
                break;
            }
        }
    }
    pnl_mat_free(&DeltaB);
    return double(count) / double(nSamples);
}

double mcam::Optim::gain_sa(int l, double gamma, bool with_averaging) const
{
    if (with_averaging) return  gamma / pow(l + 100., 0.5);
    else return gamma / (l + 100.);
}

int mcam::Optim::getIterationsSAA() const
{
#ifdef HAVE_MPI
```

```
    if (useMPI)
    {
        int size;
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        return int(std::ceil(iterationsSAA / double(size)));
    }
#endif
    return iterationsSAA;
}

/**
 * Run a Robbins Monro algorithm with truncation defined by the truncate
 * function member which can be overloaded in inherited classes to minimize
 *      E[fcost(alpha)]
 * where the classical definition for fcost(alpha) = max_i Z_i - alpha . M_i
 *
 *
 * @param rng a set of random number generators bundled in a Workspace
 * @param[in,out] alpha the coefficients of the martingale decomposition.
 * alpha must be initialized on input, so we can make use of a smart initial
 * guess if any.
 * @param outfile can be NULL. If not, the iterations of the algorithm are
 * printed in this file.
 * @param with_averaging boolean parameter. If true, alpha is averaged.
 *
 * @return
 */
int mcam::Optim::sa(mcam::PnlRng_Workspace &rng, PnlVect *alpha, FILE *outfile,
{
    const int start_averaging = iterationsSA / block_SA * 1. / 3.;
    double diameter = 0.1 * alpha->size;
    int ntrunc = 0;
    PnlVect *alpha_it;
    if (with_averaging)
    {
        alpha_it = pnl_vect_copy(alpha);
        pnl_vect_set_zero(alpha);
    }
    else
        alpha_it = alpha;
```

```
#ifdef _OPENMP
    int num_threads = std::min(omp_get_max_threads(), block_SA);
    #pragma omp parallel num_threads(num_threads)
#endif
    {
        PnlMat *DeltaB = pnl_mat_create(mart->subdates, mart->size);
        PnlVect *grad_block = NULL;
        if (block_SA > 1) grad_block = pnl_vect_create(mart->nbFreedom);
        for (int l = 0; l < iterationsSA / block_SA; l++)
        {
            const double step_gain = gain_sa(l, gamma, with_averaging) / block_S
            if (block_SA > 1) pnl_vect_set_zero(grad_block);
#ifdef _OPENMP
            #pragma omp for
#endif
            for (int k = 0; k < block_SA; k++)
            {
                pnl_mat_rng_normal(DeltaB, mart->subdates, mart->size, rng());
                mod->path(DeltaB); // sample the model
                mart->computePath(DeltaB, alpha_it);  // Compute the martingale
                dfcost();  // compute the gradient of the cost
                if (block_SA == 1)
                    grad_block = grad_m();
                else
                {
                    pnl_vect_plus_vect(grad_block, grad_m());
                }
            }
#ifdef _OPENMP
            #pragma omp critical
#endif
            {
                pnl_vect_axpby(step_gain, grad_block, 1, alpha_it);
            }
#ifdef _OPENMP
            #pragma omp master
#endif
            {
                truncate(alpha_it, diameter, ntrunc);
                if (with_averaging && (l > start_averaging)) pnl_vect_plus_vect(
                if (outfile != NULL) pnl_vect_fprint_asrow(outfile, alpha_it);
```

```
            }
        }
        pnl_mat_free(&DeltaB);
        if (block_SA > 1) pnl_vect_free(&grad_block);
    }

    if (with_averaging)
    {
        pnl_vect_div_double(alpha, (double)(iterationsSA / block_SA - start_aver
        pnl_vect_free(&alpha_it);
    }
    return ntrunc;
}

/**
 * Compute the new cost and value for the martingale decomposition given by alph
 *
 * @param p_grad [out] New value of the gradient
 * @param p_cost [out] New value of the cost
 * @param p_var [out] The variance of the SAA estimator.
 * @param p_alpha [in] coefficients of the martingale decomposition
 * @param p_DeltaB [in] array of p_iterations matrices of Brownian increments. B
 * only contains in the money samples.
 * @param p_iterations [in] number of iterations to run
 * @param moneyness probability of in the money paths
 */
void mcam::Optim::computeFullGradient(PnlVect *p_grad, double & p_cost, double &
{
    double cost = 0., var = 0.;
    pnl_vect_set_zero(p_grad);
#ifdef _OPENMP
#pragma omp parallel
#endif
    {
        PnlVect *grad_block = pnl_vect_create_from_zero(p_grad->size);
#ifdef _OPENMP
#pragma omp for reduction(+:cost)
#endif
        for (int i = 0; i < p_iterations; i++)
        {
            double cost_i;
```

```
            mod->path(p_DeltaB[i]); // sample the model
            mart->computePath(p_DeltaB[i], p_alpha);  // Compute the martingale
            cost_i = dfcost();  // compute the cost and its gradient
            cost += cost_i;
            var += cost_i * cost_i;
            pnl_vect_plus_vect(grad_block, grad_m());
        }
#ifdef _OPENMP
#pragma omp critical
#endif
        {
            pnl_vect_plus_vect(p_grad, grad_block);
        }
        pnl_vect_free(&grad_block);
    }
    pnl_vect_mult_scalar(p_grad, moneyness / p_iterations);
    p_cost = cost * moneyness / p_iterations;
    p_var = var * moneyness / p_iterations - p_cost * p_cost;
}

/**
 * Compute the new cost and value for the martingale decomposition given by alph
 * Each processor computes its contribution using its paths and the results are
 *
 * @param p_grad [out] New value of the gradient. Only relevant in the root proc
 * @param p_cost [out] New value of the cost. Only relevant in the root process.
 * @param p_var [out] The variance of the SAA estimator. Only relevant in the ro
 * @param p_alpha [in] coefficients of the martingale decomposition
 * @param p_DeltaB [in] array of p_iterations matrices of Brownian increments. B
 * only contains in the money samples.
 * @param p_iterations [in] total number of iterations to run
 * @param moneyness probability of in the money paths
 */
void mcam::Optim::computeFullGradientMPI(PnlVect *p_grad, double & p_cost, doubl
{
#ifdef HAVE_MPI
    int size, rank;
    double cost = 0., var = 0.;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
    pnl_vect_set_zero(p_grad);
    // pnl_object_mpi_bcast(PNL_OBJECT(p_alpha), 0, MPI_COMM_WORLD);
    MPI_Bcast(p_alpha->array, p_alpha->size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    for (int i = 0; i < p_iterations; i++)
    {
        double cost_i;
        mod->path(p_DeltaB[i]); // sample the model
        mart->computePath(p_DeltaB[i], p_alpha);  // Compute the martingale basi
        cost_i = dfcost();  // compute the cost and its gradient
        cost += cost_i;
        var += cost_i * cost_i;
        pnl_vect_plus_vect(p_grad, grad_m());
    }
    pnl_vect_mult_scalar(p_grad, moneyness / p_iterations);
    cost *= moneyness / p_iterations;
    var *= moneyness / p_iterations;
    MPI_Reduce(&cost, &p_cost, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    MPI_Reduce(&var, &p_var, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    pnl_object_mpi_reduce(PNL_OBJECT(p_grad), PNL_OBJECT(p_grad), MPI_SUM, 0, MP
    if (rank == 0)
    {
        p_cost /= size;
        p_var = p_var / size - p_cost * p_cost;
        pnl_vect_div_scalar(p_grad, size);
    }
#endif
}

/**
 * Sample the renormalized Brownian increments
 *
 * @param dates number of dates
 * @param size size of the Brownian motion
 * @param iterations number of sample
 * @param rng a workspace of PnlRng
 *
 * @return
 */
static PnlMat** sampleBrownianIncrements(int dates, int size, int iterations, mc
{
    PnlMat **DeltaB = new PnlMat*[iterations];
```

```
    for (int i = 0; i < iterations; i++)
    {
        DeltaB[i] = pnl_mat_create(dates, size);
        pnl_mat_rng_normal(DeltaB[i], dates, size, rng());
    }
    return DeltaB;
}

/**
 * Samples Brownian increments leading to in the money paths. It also computes t
 * price and the probability to be in the money at any of the exercising dates.
 *
 * @param iterations number of samples to run the computations
 * @param rng a workspace of PnlRng
 * @param[out] prix the price of the European option
 * @param[out] moneyness the probability to be in the money at any of the exerci
 * @param[out] nPathsInTheMoney the number of in the money paths. It is always s
 * iterations.
 *
 * @return An aray of size nPathsInTheMoney holding the Brownian increments corr
 * the money paths.
 */
PnlMat** mcam::Optim::sampleBrownianIncrementsInTheMoney(int iterations, PnlRng_
{
    PnlMat **DeltaB = new PnlMat*[iterations];
    prix = 0.;
    moneyness = 0.;
    nPathsInTheMoney = 0;
    for (int i = 0; i < iterations; i++)
    {
        bool isInTheMoney = false;
        DeltaB[nPathsInTheMoney] = pnl_mat_create(mart->subdates, mart->size);
        pnl_mat_rng_normal(DeltaB[nPathsInTheMoney], mart->subdates, mart->size,
        mod->path(DeltaB[nPathsInTheMoney]); // sample the model
        for (int t_sub = 0; t_sub <= mod->subdates; t_sub += mod->subticks)
        {
            double Zt = opt->payoff(mod->getPath(), t_sub);
            if (Zt > 0)
            {
                isInTheMoney = true;
                break;
```

```
            }
        }
        if (isInTheMoney)
        {
            nPathsInTheMoney++;
            double ZT = opt->payoff(mod->getPath(), mod->subdates);
            prix += ZT;
        }
        else
        {
            pnl_mat_free(&(DeltaB[nPathsInTheMoney]));
        }

    }
    moneyness = double(nPathsInTheMoney) / double(iterations);
    prix *= mod->discount(mod->subdates) / double(iterations);
    return DeltaB;
}

/**
 * Run a SAA method to compute the minimum of the cost function using both its
 * gradient and hessian matrix if it exists.
 *
 * @param rng a set of random number generators bundled in a Workspace.
 * @param[in,out] alpha the coefficients of the martingale decomposition.
 * alpha must be initialized on input, so we can make use of a smart initial
 * guess if any.
 * @param[out] prix the price. When used with MPI, prix is only relevant on the
 * @param stepMethod the class called to move forward from one iteration to the
 * @param useBlocks a boolean saying if we use increasing sample size.
 */
void mcam::Optim::saa(PnlRng_Workspace &rng, PnlVect *alpha, double &prix, doubl
{
    double europeanPrice, moneyness;
    PnlVect *grad_candidate = pnl_vect_create_from_zero(alpha->size);
    PnlVect *grad = pnl_vect_create_from_zero(alpha->size);
    PnlVect *alpha_candidate = pnl_vect_create_from_zero(alpha->size);
    int max_iter = 40;
    int iterations_k, block_incr;
    bool isRoot = true;
```

```
    if (useMPI)
    {
#ifdef HAVE_MPI
        int rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        isRoot = (rank == 0);
#endif
    }

    int n_iterations = getIterationsSAA();
    int nPathsInTheMoney = 0;
    PnlMat **DeltaB = sampleBrownianIncrementsInTheMoney(n_iterations, rng, euro
    if (isRoot)
    {
        std::cout << "Number of degrees of freedom: " << mart->nbFreedom << std:
        std::cout << "Moneyness: " << moneyness << std::endl;
        std::cout << "European price: " << europeanPrice << std::endl;
    }

    if (useBlocks)
    {
        block_incr = (nPathsInTheMoney / 2) / max_iter;
        iterations_k = nPathsInTheMoney / 2;
    }
    else
    {
        block_incr = 0;
        iterations_k = nPathsInTheMoney;
    }

    prix = DBL_MAX;
    var = 0.;
    double step = 1., norm = 1.;
    double cost_k = europeanPrice + 1.;
    double var_k;
    for (int k = 0; k < max_iter; k++)
    {
        int decision = 0;
        if (isRoot)
        {
            pnl_vect_clone(alpha_candidate, alpha);
```

```
            pnl_vect_axpby(step * (cost_k - europeanPrice) / norm, grad, 1, alph
        }

        if (useMPI)
            computeFullGradientMPI(grad_candidate, cost_k, var_k, alpha_candidat
        else
            computeFullGradient(grad_candidate, cost_k, var_k, alpha_candidate,

        if (isRoot)
        {
            decision = stepMethod->Step(cost_k, step, prix);
        }
#ifdef HAVE_MPI
        if (useMPI) MPI_Bcast(&decision, 1, MPI_INT, 0, MPI_COMM_WORLD);
#endif

        if (decision == Accept)
        {
            double n_samples = iterations_k / moneyness;
            if (useMPI)
            {
#ifdef HAVE_MPI
                MPI_Reduce(isRoot ? MPI_IN_PLACE : &n_samples, &n_samples, 1, MP
#endif
            }
            if (isRoot)
            {
                pnl_vect_clone(alpha, alpha_candidate);
                pnl_vect_clone(grad, grad_candidate);
                norm = pnl_vect_norm_two(grad);
                norm *= norm;
                var = var_k / n_samples;
                std::cout << "cost " << cost_k << " -- stdev " << std::sqrt(var)
            }
            iterations_k += block_incr;
        }
        else if (decision == Stop)
        {
            if (isRoot) { std::cout << "Stopping at iteration " << k << std::end
            break;
        }
```

```
    }
    pnl_vect_free(&alpha_candidate);
    pnl_vect_free(&grad_candidate);
    pnl_vect_free(&grad);

    for (int i = 0; i < nPathsInTheMoney; i++) pnl_mat_free(&(DeltaB[i]));
    delete [] DeltaB;
}

/**
 * Repeat the SAA procedure
 *
 * @param rng a set of random number generators bundled in a Workspace.
 * @param[out] alpha the coefficients of the martingale decomposition.
 * alpha must be initialized on input, so we can make use of a smart initial
 * guess if any.
 * @param stepMethod the class called to move forward from one iteration to the
 * @param iterations number of time we repeat the SAA procedure
 * @param useBlocks a boolean saying if we use increasing sample size.
 */
void mcam::Optim::saa_average(PnlRng_Workspace &rng, PnlVect *alpha, StepMethod
{

    double prix, var;
    PnlVect *alpha_i = pnl_vect_create(alpha->size);
    pnl_vect_set_zero(alpha);
    for (int i=0; i < iterations; i++)
    {
        pnl_vect_set_zero(alpha_i);
        saa(rng, alpha_i, prix, var, stepMethod, useBlocks);
        std::cout << "Price (SAA): " << prix << std::endl;
        std::cout << "Standard deviation (SAA): " << std::sqrt(var) << std::endl
        pnl_vect_plus_vect(alpha, alpha_i);
    }
    pnl_vect_div_scalar(alpha, iterations);
    pnl_vect_free(&alpha_i);
}

/**
 * Run a Monte Carlo algorithm to compute the gradient of the cost
 *
```

```
 * @param rng a set of random number generators bundled in a Workspace
 * @param alpha the optimal solution of the minimization problem
 * @param[out] E contains |E[dfcost]|_infty on output
 * @param[out] p_prix contains E[cost]
 * @param[out] p_var variance for the estimator of E[cost]
 * @param ComputeDf boolean parameter. If false, only compute E[cost], \ a E is
 * left unassigned
 */
void mcam::Optim::EGradCost(PnlRng_Workspace &rng, const PnlVect *alpha, double
{
    double prix = 0., var = 0.;
    PnlVect *grad = NULL;
    E = 0.;
    grad = pnl_vect_create_from_zero(mart->nbFreedom);
    /*
     * Monte Carlo loop
     */
#ifdef _OPENMP
    #pragma omp parallel
    {
#endif
        PnlMat *DeltaB = pnl_mat_create(mart->subdates, mart->size);
        PnlVect *grad_thread = NULL;
        if (ComputeDf) grad_thread = pnl_vect_create_from_zero(mart->nbFreedom);

        #pragma omp for reduction(+:prix) reduction(+:var)
        for (int l = 0; l < iterationsMC; l++)
        {
            double tmp;
            pnl_mat_rng_normal(DeltaB, mart->subdates, mart->size, rng());
            mod->path(DeltaB); // sample the model
            mart->computePath(DeltaB, alpha);
            tmp = dfcost(false);  // compute the cost
            prix += tmp;
            var += tmp * tmp;
            if (ComputeDf)
            {
                dfcost(true);  // compute the gradient
                pnl_vect_plus_vect(grad_thread, grad_m());
            }
        }
```

```
#ifdef _OPENMP
        #pragma omp critical
        {
            if (ComputeDf) pnl_vect_plus_vect(grad, grad_thread);
        }
#endif
        pnl_vect_free(&grad_thread);
        pnl_mat_free(&DeltaB);
#ifdef _OPENMP
    }
#endif
    p_prix = prix / iterationsMC;
    p_var = var / iterationsMC - p_prix * p_prix;
    p_var /= iterationsMC;
    if (ComputeDf)
    {
        pnl_vect_div_double(grad, iterationsMC);
        E = pnl_vect_norm_infty(grad);
    }
    pnl_vect_free(&grad);
}

mcam::MaxCost::MaxCost() : Optim() { }

mcam::MaxCost::MaxCost(Model *mod, Option *opt, Martingale *mart, const Param &P
    : Optim(mod, opt, mart, P) { }

mcam::MaxCost::~MaxCost() { }

void mcam::MaxCost::print() const
{
    mcam::Optim::print();
    std::cout << "******************************" << std::endl;
}

/**
 * Compute the subgradient of the cost and store it into grad_m.
 * Return the cost value
 *
 * This cost is just convex.
 *
```

```
 * @param ComputeDf boolean parameter. If false, only return the cost and do
 * not compute the gradient
 * @return the cost value
 */
double mcam::MaxCost::dfcost(bool ComputeDf)
{
    if (ComputeDf)
    {
        pnl_vect_resize(grad_m(), mart->nbFreedom);
        pnl_vect_set_zero(grad_m());
    }
    double sup = opt->payoff(mod->getPath(), 0);
    bool isInTheMonney = (sup > 0);
    int tau = mod->subdates; // It needs no initialization
    for (int t = 1, t_sub = 0; t < mod->dates + 1; t++)
    {
        t_sub += mod->subticks;
        double Zt = opt->payoff(mod->getPath(), t_sub) * mod->discount(t_sub);
        double hat_Mt = 0.;
        if (isInTheMonney)
        {
            double Mt = mart->at(t_sub);
            double Mtau = mart->at(tau);
            hat_Mt = Mt - Mtau;
        }
        double current_cost = Zt - hat_Mt;

        if (!isInTheMonney && (Zt > 0))
        {
            tau = t_sub;
            isInTheMonney = true;
        }
        if (current_cost > sup)
        {
            sup = current_cost;
            if (ComputeDf)
            {
                PnlVect GradMt = mart->gradat(t_sub);
                PnlVect GradMtau = mart->gradat(tau);
                pnl_vect_clone(grad_m(), &GradMt);
                pnl_vect_minus_vect(grad_m(), &GradMtau);
```

```
            }
        }
    }
    return sup;
}

void mcam::MaxCost::truncate(PnlVect *alpha, double &diameter, int &ntrunc)
{
    return;
    if (pnl_vect_norm_two(alpha) > diameter)
    {
        ntrunc ++;
        pnl_vect_set_zero(alpha);
        diameter = alpha ->size * std::log(1 + ntrunc);
    }
}
```