Help

```
#include <iostream>
#include <fstream>
#include <cmath>
#include <cstdlib>

#include "
href../../../../common/math/mcam/src/MonteCarlo_h_src.pdfMonteCarlo.hpp"
#ifdef HAVE_MPI
#include "pnl/pnl_mpi.h"
#endif

using namespace std;

#if 0  /* These functions are redundant with calling mcam::MaxCost::EGradCost */
/**
 * Compute the forward price of the American option using the already computed
 * dual decomposition
 *
 * @param[out] p_prix contains the forward price on output
 * @param[out] p_var contains the variance of the dual price on output
 * @param p_opt Option
 * @param p_mod Model
 * @param p_mart martingale
 * @param p_alpha martingale weights
 * @param p_DeltaB an array of samples for the increment of the Brownian motion
 * @param p_lengthDeltaB number of samples in p_DeltaB
 */
void forward_price(double &p_prix, double &p_var, Option *p_opt, Model *p_mod, M
                   const PnlVect *p_alpha, PnlMat **p_DeltaB, int p_lengthDeltaB
{
    double prix = 0.;
    double var = 0.;

#ifdef _OPENMP
#pragma omp parallel for reduction(+:prix) reduction(+:var)
#endif
    for (int l = 0; l < p_lengthDeltaB; l++)
    {
        p_mod->path(p_DeltaB[l]);
```

```
        p_mart->computePath(p_DeltaB[l], p_alpha);
        double payoff_tau = 0.;
        double max_l = 0.;
        for (int t = 0; t < p_mod->dates + 1; t++)
        {
            int t_sub = t * p_mod->subticks;
            double Mt = p_mart->at(t_sub);
            double payoff = p_opt->payoff(p_mod->getPath(), t_sub) * p_mod->disc
            if (payoff - Mt > max_l)
            {
                payoff_tau = payoff;
                max_l = payoff - Mt;
            }
        }
        prix += payoff_tau;
        var += payoff_tau * payoff_tau;
    }
    prix = prix / p_lengthDeltaB;
    var = var / p_lengthDeltaB - prix * prix;
    p_prix = prix;
    p_var = var;
}


/**
 * Compute the forward price of the American option using the already computed
 * dual decomposition
 *
 * @param[out] p_prix contains the forward price on output
 * @param[out] p_var contains the variance of the dual price on output
 * @param p_opt Option
 * @param p_mod Model
 * @param p_mart martingale
 * @param p_alpha martingale weights
 * @param p_iterationsMC number of Monte Carlo samples
 * @param p_rng random number genrator
 */
void forward_price(double &p_prix, double &p_var, Option *p_opt, Model *p_mod, M
                   const PnlVect *p_alpha, int p_iterationsMC, PnlRng_Workspace
{
    double prix = 0.;
```

```
    double var = 0.;


#ifdef _OPENMP
    #pragma omp parallel
#endif
    {
        PnlMat *DeltaB = pnl_mat_create(p_mart->subdates, p_mart->size);
#ifdef _OPENMP
        #pragma omp for reduction(+:prix) reduction(+:var)
#endif
        for (int l = 0; l < p_iterationsMC; l++)
        {
            pnl_mat_rng_normal(DeltaB, p_mart->subdates, p_mart->size, p_rng());
            p_mod->path(DeltaB);
            p_mart->computePath(DeltaB, p_alpha);
            double payoff_tau = 0.;
            double max_l = 0.;
            for (int t = 0; t < p_mod->dates + 1; t++)
            {
                int t_sub = t * p_mod->subticks;
                double Mt = p_mart->at(t_sub);
                double payoff = p_opt->payoff(p_mod->getPath(), t_sub) * p_mod->
                if (payoff - Mt > max_l)
                {
                    payoff_tau = payoff;
                    max_l = payoff - Mt;
                }
            }
            prix += payoff_tau;
            var += payoff_tau * payoff_tau;
        }
        pnl_mat_free(&DeltaB);
    }
    prix = prix / p_iterationsMC;
    var = var / p_iterationsMC - prix * prix;
    p_prix = prix;
    p_var = var;
}
#endif
```

```
#if 0
/**
 * Compute the backward price of the American option using the already computed
 * dual decomposition. This is based on the dynamic programming equation in
 * which E[U_{t+1} | F_t] is approximated by U_{t+1} - M_{t+1} + M_t.
 *
 * Note that because of M is not optimal, the policy obtained by this
 * algorithm is not a stopping time and hence we cannot use M as a control
 * variate.
 *
 * @warning The results are not good.
 *
 * @param[out] p_prix contains the backward price on output
 * @param[out] p_var contains the variance of the dual price on output
 * @param p_opt Option
 * @param p_mod Model
 * @param p_mart martingale
 * @param p_alpha martingale weights
 * @param p_iterationsMC number of Monte Carlo samples
 * @param p_rng random number genrator
 */
void backward_price(double &p_prix, double &p_var, Option *p_opt, Model *p_mod,
                    const PnlVect *p_alpha, int p_iterationsMC, PnlRng_Workspace
{
    int T = p_mart->subdates;
    double prix = 0.;
    double var = 0.;

#ifdef _OPENMP
    #pragma omp parallel
#endif
    {
        PnlMat *DeltaB = pnl_mat_create(T, p_mart->size);
#ifdef _OPENMP
        #pragma omp for reduction(+:prix) reduction(+:var)
#endif
        for (int l = 0; l < p_iterationsMC; l++)
        {
            PnlVect St;
            double prix_l = 0.;
            double Mt = 0., Ut = 0.;
```

```
            double Mt_next, Ut_next;

            pnl_mat_rng_normal(DeltaB, T, p_mart->size, p_rng());
            p_mod->path(DeltaB);
            p_mart->computePath(DeltaB, p_alpha);
            Mt = p_mart->at(T);
            St = pnl_vect_wrap_mat_row(p_mod->getPath(), T);
            Ut = p_opt->payoff(&St, T * p_mod->dt) * p_mod->discount(T);
            prix_l = Ut;
            for (int t = T - 1; t >= 0; t--)
            {
                Ut_next = Ut;
                Mt_next = Mt;
                Mt = (0. ? (t == 0) : p_mart->at(t));
                St = pnl_vect_wrap_mat_row(p_mod->getPath(), t);
                double payoff = p_opt->payoff(&St, t * p_mod->dt) * p_mod->disco
                double tmp = Mt - Mt_next + Ut_next;
                if (payoff >= tmp && payoff > 0)
                {
                    prix_l = payoff;
                    Ut = payoff;
                }
                else
                    Ut = tmp;
            }
            prix += prix_l;
            var += prix_l * prix_l;
        }
        pnl_mat_free(&DeltaB);
    }
    prix = prix / p_iterationsMC;
    var = var / p_iterationsMC - prix * prix;
    p_prix = prix;
    p_var = var;
}
#endif

#if 0
/**
 * Compute the backward price of the American option using the already computed
 * dual decomposition and the dual price. This is based on the computation of
```

```
 * the largest optimal stopping time
 *  sup{t : Z_t >= U_0 + M_t} = sup{t : A_t = 0}
 *
 * Note that because M is not optimal, the policy obtained by this
 * algorithm is not a stopping time and hence we cannot use M as a control
 * variate.
 *
 * @warning This method does not give good results.
 *
 * @param[inout] p_prix contains the dual price on input and the backward price
 * @param[out] p_var contains the variance of the dual price on output
 * @param p_opt Option
 * @param p_mod Model
 * @param p_mart martingale
 * @param p_alpha martingale weights
 * @param p_iterationsMC number of Monte Carlo samples
 * @param p_rng random number genrator
 */
void backward_price_At(double &p_prix, double &p_var, Option *p_opt, Model *p_mo
                       const PnlVect *p_alpha, int p_iterationsMC, PnlRng_Worksp
{
    int T = p_mart->subdates;
    double U0 = p_prix;
    double prix = 0.;
    double var = 0.;

#ifdef _OPENMP
    #pragma omp parallel
#endif
    {
        PnlMat *DeltaB = pnl_mat_create(T, p_mart->size);
#ifdef _OPENMP
        #pragma omp for reduction(+:prix) reduction(+:var)
#endif
        for (int l = 0; l < p_iterationsMC; l++)
        {
            PnlVect St;
            double payoff;
            double prix_l = 0.;
            double Mt = 0.;
```

```
            pnl_mat_rng_normal(DeltaB, T, p_mart->size, p_rng());
            p_mod->path(DeltaB);
            p_mart->computePath(DeltaB, p_alpha);
            Mt = p_mart->at(T);
            St = pnl_vect_wrap_mat_row(p_mod->getPath(), T);
            prix_l = p_opt->payoff(&St, T * p_mod->dt) * p_mod->discount(T);
            for (int t = T - 1; t >= 0; t--)
            {
                Mt = (0. ? (t == 0) : p_mart->at(t));
                St = pnl_vect_wrap_mat_row(p_mod->getPath(), t);
                payoff = p_opt->payoff(&St, t * p_mod->dt) * p_mod->discount(t);
                double tmp = U0 + Mt;
                if (payoff >= tmp && payoff > 0)
                {
                    prix_l = payoff;
                    break;
                }
            }
            prix += prix_l;
            var += prix_l * prix_l;
        }
        pnl_mat_free(&DeltaB);
    }
    p_prix = p_prix / p_iterationsMC;
    p_var = p_var / p_iterationsMC - p_prix * p_prix;
    prix = p_prix;
    var = p_var;
}
#endif

mcam::MonteCarlo::MonteCarlo(Model *p_mod, Option *p_opt, Martingale *p_mart, co
    : mod(p_mod), opt(p_opt), mart(p_mart)
{
    regressionTypeString = p_regressionTypeString;
    if (regressionTypeString == "pol")
    {
        regType = REG_POL;
        p_map.extract("degree for polynomial regression", regressionDegree);
    }
    else if (regressionTypeString == "chaos")
    {
```

```
        regType = REG_CHAOS;
        p_map.extract("degree for chaos regression", regressionDegree);
    }
    else if (regressionTypeString == "reduced_chaos")
    {
        regType = REG_REDUCED_CHAOS;
        p_map.extract("degree for chaos regression", regressionDegree);
    }
    else
    {
        std::cout << "Value " << regressionTypeString << " is unknown." << std::
        abort();
    }
    p_map.extract("MC iterations", iterationsMC);

}

void mcam::MonteCarlo::print() const
{
    std::cout << "Monte Carlo samples: " << iterationsMC << "\ n";
    std::cout << "Regression type: " << regressionTypeString << "\ n";
    std::cout << "Regression degree: " << regressionDegree << "\ n";
}
/**
 * Solve the backward iterations
 *
 * @param p_opt Option
 * @param p_mod Model
 * @param p_reg the regressor
 * @param p_Z the array of payoffs (the length of the array is the number of exe
 * @param p_Ztau the optimal value of the payoff
 * @param p_tau the optimal times
 * @param p_iterationsMC number of MC samples
 * @param useMPI true if you are running through MPI.
 */
void mcam::MonteCarlo::solve_backward_induction(mcam::Regression *p_reg, PnlVect
{
    PnlVect *beta = pnl_vect_new();
    int T = mod->subdates;
    // Backward step
    for (int t = T - mod->subticks, n = mod->dates - 1; t > 0; t -= mod->subtick
```

```
    {
        p_reg->setCurrentDate(t);
        p_reg->computeCoefficients(beta, p_Ztau, p_Z[n]);

#ifdef HAVE_MPI
        if (p_useMPI)
        {
            int size;
            MPI_Comm_size(MPI_COMM_WORLD, &size);
            pnl_object_mpi_reduce(PNL_OBJECT(beta), PNL_OBJECT(beta), MPI_SUM, 0
            pnl_vect_div_scalar(beta, size);
            pnl_object_mpi_bcast(PNL_OBJECT(beta), 0, MPI_COMM_WORLD);
        }
#endif

        // Update each path
        for (int l = 0; l < p_iterationsMC; l++)
        {
            double payoff = GET(p_Z[n], l);
            if (payoff == 0.) continue;
            double Et = p_reg->computeRegressorValue(l, beta);
            if (payoff > 0. && payoff > Et)
            {
                LET(p_Ztau, l) = payoff;
                LET_INT(p_tau, l) = t;
            }
        }
        // Compute variance at time 2
        // if (t == 2 * mod->subticks)
        // {
        //     double prix = 0., var = 0.;
        //     for (int l = 0; l < iterationsMC; l++)
        //     {
        //         double payoff = GET(p_Ztau, l);
        //         prix += payoff;
        //         var += payoff * payoff;
        //     }
        //     prix /= iterationsMC;
        //     var = var / iterationsMC - prix * prix;
        //     std::cout << "Variance at time 2: " << var << "\ n";
        // }
```

```
    }
    pnl_vect_free(&beta);
}

static int getIterationsMC(int p_iterationsMC, bool p_useMPI)
{
#ifdef _HAVE_MPI
    if (p_useMPI)
    {
        int size;
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        return int(std::ceil(p_iterationsMC / double(size)));
    }
#endif
    return p_iterationsMC;
}


/**
 * Dynamic propgramming principle with control variate. If alpha is set to 0, no
 *
 * @param[out] p_prix contains the backward price on output
 * @param[out] p_var contains the variance of the dual price on output
 * @param opt Option
 * @param mod Model
 * @param p_dual martingale. It is only for using the martingale as a control va
 * @param p_alpha martingale weights. If we do not use any control, set p_alpha
 * @param p_rng random number genrator
 * @param iterationsMC number of MC samples
 * @param p_type the type of regression to use (REG_POL, REG_CHAOS, REG_REDUCED_
 */
void mcam::MonteCarlo::backward_price_dpp(double &p_prix, double &p_var, const P
{
    int tmp = 0;
#ifdef HAVE_MPI
    MPI_Initialized(&tmp);
#endif
    bool useMPI = (tmp == 1);
    bool isRoot = true;
#ifdef HAVE_MPI
    if (useMPI)
```

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

    PnlMat **DeltaB = new PnlMat*[iterationsMC];
    PnlMat **regressors = new PnlMat*[iterationsMC];
    PnlVect **Z = new PnlVect*[mod->dates + 1];
    PnlVect *Ztau = pnl_vect_create(iterationsMC);
    PnlVectInt *tau = pnl_vect_int_create(iterationsMC);
    int T = mod->subdates;
    double prix = 0., var = 0.;
    Regression *reg;
    int n_iterationsMC = getIterationsMC(iterationsMC, useMPI);

    pnl_vect_int_set_int(tau, T);

    for (int i = 0; i < mod->dates + 1; i++)
    {
        Z[i] = pnl_vect_create(iterationsMC);
    }
    for (int l = 0; l < n_iterationsMC; l++)
    {
        DeltaB[l] = pnl_mat_create(T, mod->brownianSize);
        pnl_mat_rng_normal(DeltaB[l], T, mod->brownianSize, p_rng());
        mod->path(DeltaB[l]);
        regressors[l] = pnl_mat_copy(mod->getRegressor());
        for (int t = 0, i = 0; t < T + 1; t += mod->subticks, i++)
        {
            LET(Z[i], l) = opt->payoff(mod->getPath(), t) * mod->discount(t);
        }
        LET(Ztau, l) = GET(Z[mod->dates], l);
    }

    switch (regType)
    {
    case REG_POL:
    {
        reg = new RegressionPol(mod->brownianSize, regressionDegree, regressors,
```

```
        RegressionPol &reg_pol = dynamic_cast<RegressionPol&>(*reg);
        reg_pol.setReduced(mod);
    }
        break;
    case REG_CHAOS:
        reg = new RegressionChaos(mod->brownianSize, regressionDegree, DeltaB, n
        break;
    case REG_REDUCED_CHAOS:
        reg = new RegressionChaosReduced(mod->brownianSize, regressionDegree, De
        break;
    default:
        abort();
        break;
    }
    solve_backward_induction(reg, Z, Ztau, tau, n_iterationsMC, useMPI);
    delete reg;

    // Monte Carlo
    for (int i = 0; i < n_iterationsMC; i++)
    {
        double payoff = GET(Ztau, i);
        if (mart && p_alpha)
        {
            int tau_i = GET_INT(tau, i);
            mart->computePath(DeltaB[i], p_alpha);
            double Mt = mart->at(tau_i);
            payoff -= Mt;
        }
        prix += payoff;
        var += payoff * payoff;
    }

    prix /= n_iterationsMC;
    var = var / n_iterationsMC;
    if (useMPI)
    {
#ifdef HAVE_MPI
        int size;
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        MPI_Reduce(&prix, &p_prix, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
        MPI_Reduce(&var, &p_var, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

```
        p_prix /= size;
        p_var = (p_var / size - p_prix * p_prix) / (size * n_iterationsMC);
#endif
    }
    else
    {
        p_prix = prix;
        p_var = (var - prix * prix) / n_iterationsMC;
    }

    if (isRoot)
    {
        // Exercise at time 0?
        // At time 0, only the spot is used to compute the payoff so we can use
        double prix_0 = GET(Z[0], 0);
        if (prix_0 > p_prix)
        {
            p_prix = prix_0;
            p_var = 0.;
        }
    }

    pnl_vect_int_free(&tau);
    pnl_vect_free(&Ztau);
    for (int i = 0; i < n_iterationsMC; i++)
    {
        pnl_mat_free(&DeltaB[i]);
        pnl_mat_free(&regressors[i]);
    }
    for (int i = 0; i < mod->dates + 1; i++)
    {
        pnl_vect_free(&Z[i]);
    }

    delete [] DeltaB;
    delete [] Z;
    delete [] regressors;
}
```