

## Help

```

#ifndef _MULTILEVELMONTECARLO_HPP
#define _MULTILEVELMONTECARLO_HPP

#include "Model.hpp"
#include "Option.hpp"
#include "MonteCarlo.hpp"
#include "MonteCarloMode.hpp"
#include "pnl/pnl_matrix.h"
#include "pnl/pnl_random.h"
#include "parser.hpp"

template <class Mode, class MultiLevelMode> class MultiLevelMonteCarlo
{
public:
    PnlRng *rng;
    Param ParamTab;

    MultiLevelMonteCarlo() : rng(NULL) { }

    MultiLevelMonteCarlo(PnlRng *r, const Param &P) :
        rng(r), ParamTab(P)
    {
        P.extract("option size", modelSize);
        P.extract("timestep number finest level", numberOfStepsFinestLevel);
        P.extract("timestep number level one", numberOfStepsLevel1);
        P.extract("maturity", maturity);

        numberOfLevels = int(pnl_round(std::log(numberOfStepsFinestLevel) / std::log
        ParamTab.insert("timestep number", T_INT, numberOfTimeSteps(0));
        ParamTab.insert("sample number", T_LONG, numberOfSamples(0));
        ParamTab.insert("sample number SAA", T_LONG, numberOfSamplesSAA(0));
    }

    ~MultiLevelMonteCarlo() { ParamTab.~Param(); }

    void print()
    {
        std::cout << std::endl;
        std::cout << "*****" << std::endl;
    }

```

```

        std::cout << "**** Monte Carlo Characteristics ****" << std::endl;
        std::cout << " MultiLevel" << std::endl;
        std::cout << " Number of levels : " << numberOfLevels << std::endl;
        std::cout << " Number of time steps level one : " << numberOfStepsLevel1 <
        std::cout << "*****" << std::endl << std:::
    }

/**
 * Return the number of samples to use in level
 *
 * @param level the index of the current level
 *
 * @return an integer
 */
size_t numberOfSamples(int level) const
{
    size_t n = size_t(numberOfLevels * maturity * (numberOfStepsLevel1 - 1)
        * pnl_pow_i(double(numberOfStepsLevel1), 2 * numberOfLevels -
    return n;
}

/**
 * Return the number of samples to use in level for SAA
 *
 * @param level the index of the current level
 *
 * @return an integer
 */
size_t numberOfSamplesSAA(int level) const
{
    size_t N_l = numberOfSamples(level);
    size_t m_l = numberOfTimeSteps(level);
    size_t K_Newton = 5; // maximum number of Newton's iterations
    return size_t(N_l * m_l / (m_l + 3 * K_Newton));
}

/**
 * Return the number of samples to use in level
 *
 * @param level the index of the current level
 *

```

```

    * @return an integer
    */
int numberOfTimeSteps(int level) const
{
    return pnl_pow_i(numberOfStepsLevel1, level);
}

/**
 * Computes the price using a multi level Monte Carlo estimator
 *
 * @param[out] prix price
 * @param[out] std_dev of the ML estimator
 */
void Price(double &prix, double &std_dev)
{
    prix = 0.;
    double var = 0.;
    std_dev = 0.;
    for (int l = 0; l <= numberOfLevels; l++)
    {
        double prix_l, std_dev_l;
        if (l == 0)
        {
            MonteCarloCrude<Mode> mc_0(rng, ParamTab);
            mc_0.Price(prix_l, std_dev_l);
        }
        else
        {
            MonteCarloCrude<MultiLevelMode> *mc = InitLevel(l);
            mc->Price(prix_l, std_dev_l);
            delete mc;
        }
        std::cout << "Level " << l << " price " << prix_l << std::endl;
        std::cout << "Level " << l << " var " << std_dev_l * std_dev_l << std::endl;
        prix += prix_l;
        var += std_dev_l * std_dev_l;
    }
    std_dev = std::sqrt(var);
}

/**

```

```

* Computes the price and delta using a multi level Monte Carlo estimator
*
* @param[out] prix price
* @param[out] std_devprix std_dev of the ML estimator
* @param[out] delta delta of the option
* @param[out] std_devdelta std_dev of the FD ML estimator
*/
void PriceDelta(double &prix, double &std_devprix, PnlVect *delta, PnlVect *std_devdelta)
{
    PnlVect *delta_1, *std_devdelta_1;
    prix = 0.;
    std_devprix = 0.;
    pnl_vect_set_zero(delta);
    pnl_vect_set_zero(std_devdelta);
    delta_1 = pnl_vect_create(modelSize);
    std_devdelta_1 = pnl_vect_create(modelSize);
    for (int l = 0; l <= numberOfLevels; l++)
    {
        double prix_l, std_dev_l;
        if (l == 0)
        {
            MonteCarloCrude<Mode> mc(rng, ParamTab);
            mc.PriceDelta(prix_l, std_dev_l, delta_1, std_devdelta_1);
        }
        else
        {
            MonteCarloCrude<MultiLevelMode> *mc = InitLevel(l);
            mc->PriceDelta(prix_l, std_dev_l, delta_1, std_devdelta_1);
            delete mc;
        }
        std::cout << "Level " << l << " price " << prix_l << std::endl;
        std::cout << "Level " << l << " var " << std_dev_l * std_dev_l << std::endl;
        prix += prix_l;
        std_devprix += std_dev_l * std_dev_l;
        pnl_vect_plus_vect(delta, delta_1);
        pnl_vect_mult_vect_term(std_devdelta_1, std_devdelta_1);
        pnl_vect_plus_vect(std_devdelta, std_devdelta_1);
    }
    std_devprix = std::sqrt(std_devprix);
    pnl_vect_map_inplace(std_devdelta, std::sqrt);
    pnl_vect_free(&delta_1);
}

```

```

    pnl_vect_free(&std_devdelta_l);
}

/**
 * Computes the price using a sample average based IS ML estimator
 *
 * @param[out] theta computed drift
 * @param[out] prix estimated price
 * @param[out] std_dev asymptotic std_dev of the estimator
 * @param verbose a boolean to activate message printing
 */
void mc_sample_averaging(PnlVect *theta, double &prix, double &std_dev, bool v
{
    prix = 0.;
    std_dev = 0.;
    for (int l = 0; l <= numberOfLevels; l++)
    {
        double prix_l, std_dev_l;
        if (l == 0)
        {
            MonteCarloHelper<Mode> mc(rng, ParamTab);
            mc.mc_sample_averaging(theta, prix_l, std_dev_l, !verbose);
        }
        else
        {
            MonteCarloHelper<MultiLevelMode> *mc = InitLevel(l);
            mc->mc_sample_averaging(theta, prix_l, std_dev_l, !verbose);
            delete mc;
        }
        if (verbose)
        {
            std::cout << "Theta Level " << l << ": ";
            pnl_vect_print_asrow(theta);
        }
        std::cout << "Level " << l << " price " << prix_l << std::endl;
        std::cout << "Level " << l << " var " << std_dev_l * std_dev_l << std::endl;
        prix += prix_l;
        std_dev += std_dev_l * std_dev_l;
    }
    std_dev = std::sqrt(std_dev);
}

```

```

/**
 * Computes the price and deltas using a sample average based IS ML estimator
 *
 * @param[out] theta computed drift
 * @param[out] prix estimated price
 * @param[out] std_dev asymptotic std_dev of the estimator
 * @param[out] delta vector of the deltas
 * @param[out] std_devdelta std_dev of the deltas
 * @param verbose a boolean to activate message printing
 */
void mc_sample_averaging_delta(PnlVect *theta, double &prix, double &std_dev,
{
    PnlVect *delta_l, *std_devdelta_l;
    prix = 0.;
    std_dev = 0.;
    pnl_vect_set_zero(delta);
    pnl_vect_set_zero(std_devdelta);
    delta_l = pnl_vect_create(modelSize);
    std_devdelta_l = pnl_vect_create(modelSize);
    for (int l = 0; l <= numberOfLevels; l++)
    {
        double prix_l, std_dev_l;
        if (l == 0)
        {
            MonteCarloHelper<Mode> mc(rng, ParamTab);
            mc.mc_sample_averaging_delta(theta, prix_l, std_dev_l, delta_l, s
        }
        else
        {
            MonteCarloHelper<MultiLevelMode> *mc = InitLevel(l);
            mc->mc_sample_averaging_delta(theta, prix_l, std_dev_l, delta_l, s
            delete mc;
        }

        if (verbose)
        {
            std::cout << "Theta Level " << l << ": ";
            pnl_vect_print_asrow(theta);
            std::cout << "Level " << l << " price " << prix_l << std::endl;
            std::cout << "Level " << l << " var " << std_dev_l * std_dev_l <<

```

```

        }
        prix += prix_l;
        std_dev += std_dev_l * std_dev_l;
        pnl_vect_plus_vect(delta, delta_l);
        pnl_vect_mult_vect_term(std_devdelta_l, std_devdelta_l);
        pnl_vect_plus_vect(std_devdelta, std_devdelta_l);
    }
    std_dev = std::sqrt(std_dev);
    pnl_vect_map_inplace(std_devdelta, std::sqrt);
    pnl_vect_free(&delta_l);
    pnl_vect_free(&std_devdelta_l);
}

protected:
    int numberOfStepsFinestLevel; /// Number of time steps for the finest level
    int numberOfLevels; /// Number of Levels
    int modelSize;
    int numberOfStepsLevel1; /// Number of steps for Level 1
    double maturity;

    /**
     * Create a MonteCarlo object corresponding to level
     *
     * @param level the index of the current level
     *
     * @return
     */
    MonteCarloHelper<MultiLevelMode>* InitLevel(int level)
    {
        Param P(ParamTab);
        P.set("sample number", numberOfSamples(level));
        P.set("sample number SAA", numberOfSamplesSAA(level));
        P.set("timestep number", numberOfTimeSteps(level));
        MonteCarloHelper<MultiLevelMode> *mc = new MonteCarloHelper<MultiLevelMode>(P);
        return mc;
    }

};

#endif /* _MULTILEVELMONTECARLO_HPP */

```