

Help

```

#ifndef _MONTECARLO_H
#define _MONTECARLO_H

#include "math/ImportanceSampling_jl/src/Model.hpp"
#include "math/ImportanceSampling_jl/src/Option.hpp"
#include "pnl/pnl_matrix.h"
#include "pnl/pnl_random.h"
#include "math/ImportanceSampling_jl/src/parser.hpp"

#define NUMBER_OF_SAMPLES_SAA_MAX 5E5

/**
 * Base abstract class for Monte Carlo
 */
class MonteCarloBase
{
protected:
    double step;    /*!< step for the stochastic algorithm */
    size_t numberOfSamples;    /*!< number of simulations */

public:
    PnlRng *rng;
    double fdStep;    /*!< Finite difference step size */

    MonteCarloBase();
    MonteCarloBase(PnlRng *, const Param &ParamTab);
    virtual ~MonteCarloBase();
    inline size_t& getNumberOfSamples() { return numberOfSamples; }

    virtual void print(bool verbose=false) const = 0;

    virtual void Price(double &prix, double &var) = 0;
    virtual void Price_evol(const char *price_file, double &prix, double &var) = 0;
    virtual void PriceDelta(double &prix, double &varprix, PnlVect *delta, PnlVect
    virtual void expectation_order_n(PnlVect *const *g, const PnlVect *theta, cons

    virtual void RM_routine_coupled(const char *file_name, PnlVect *theta, double
    virtual void RM_routine(const char *file_name, PnlVect *theta, double &price,

```

```

virtual void RM_routine_average(const char *drift_file, const char *price_file
virtual void RM_routine_average_coupled(const char *drift_file, const char *pr
virtual void sample_averaging_newton(PnlVect *theta, bool noverbose) = 0;
virtual void mc_sample_averaging(PnlVect *theta, double &price, double &var, b
virtual void mc_sample_averaging_delta(PnlVect *theta, double &price, double &
virtual void sample_averaging_newton_poisson(PnlVect *mu, bool noverbose) = 0;
virtual void mc_sample_averaging_poisson(PnlVect *mu, double &price, double &v
virtual void sample_averaging_newton_gaussian_poisson(PnlVect *theta, PnlVect
virtual void mc_sample_averaging_gaussian_poisson(PnlVect *theta, PnlVect *mu,

};

/**
 * Class for running crude Monte Carlo
 */
template<class Mode> class MonteCarloCrude: public MonteCarloBase
{
public:
    Mode CurrentMode;

    MonteCarloCrude();
    MonteCarloCrude(PnlRng *, const Param &ParamTab);
    virtual ~MonteCarloCrude();
    virtual void print(bool verbose=false) const;

    virtual void Price(double &prix, double &var);
    virtual void Price_evol(const char *price_file, double &prix, double &var);
    virtual void PriceDelta(double &prix, double &varprix, PnlVect *delta, PnlVect

    // Unimplemented method at that level
    virtual void expectation_order_n(PnlVect *const *g, const PnlVect *theta, cons
    virtual void RM_routine_coupled(const char *file_name, PnlVect *theta, double
    virtual void RM_routine(const char *file_name, PnlVect *theta, double &price,
    virtual void RM_routine_average(const char *drift_file, const char *price_file
    virtual void RM_routine_average_coupled(const char *drift_file, const char *pr
    virtual void sample_averaging_newton(PnlVect *theta, bool noverbose) { }
    virtual void mc_sample_averaging(PnlVect *theta, double &price, double &var, b
    virtual void mc_sample_averaging_delta(PnlVect *theta, double &price, double &
    virtual void sample_averaging_newton_poisson(PnlVect *mu, bool noverbose) { }
    virtual void mc_sample_averaging_poisson(PnlVect *mu, double &price, double &v
    virtual void sample_averaging_newton_gaussian_poisson(PnlVect *theta, PnlVect

```

```

    virtual void mc_sample_averaging_gaussian_poisson(PnlVect *theta, PnlVect *mu,

protected:
    /**
     * Reduce the price and the variance
     *
     * @param[in, out] price
     * @param[in, out] var On output, it contains the variance of the estimator
     * @param Nsamples number of samples used in the estimator
     */
    void finalize(double &price, double &var, size_t Nsamples) const;
    /**
     * Resize and set to zero
     *
     * @param[in, out] delta vector
     * @param[in, out] vardelta vector
     */
    void prepare_delta(PnlVect *delta, PnlVect *vardelta);
    /**
     * Finalize the computation of the delta's and vardelta's
     *
     * @param[in, out] delta vector
     * @param[in, out] vardelta vector
     */
    void finalize_delta(PnlVect *delta, PnlVect *vardelta);

protected:
    using MonteCarloBase::rng;
    using MonteCarloBase::numberOfSamples;
    using MonteCarloBase::step;
    using MonteCarloBase::fdStep;
    using MonteCarloBase::expectation_order_n;
};

/**
 * Class used to implement numerical routines: Monte Carlo
 * and all the different stochastic algorithms. The kind of
 * importance sampling FULL or REDUCED is parametrized by
 * the template \ <Mode\ >.
 */

```

```

template<class Mode> class MonteCarloHelper: public MonteCarloCrude<Mode>
{
public:
    MonteCarloHelper();
    MonteCarloHelper(PnlRng *rng, const Param &ParamTab);
    ~MonteCarloHelper();
    void print(bool verbose=false) const;

    virtual void expectation_order_n(PnlVect *const *g, const PnlVect *theta, const
    virtual void RM_routine_coupled(const char *file_name, PnlVect *theta, double
    virtual void RM_routine(const char *file_name, PnlVect *theta, double &price,
    virtual void RM_routine_average(const char *drift_file, const char *price_file
    virtual void RM_routine_average_coupled(const char *drift_file, const char *pr
    virtual void sample_averaging_newton(PnlVect *theta, bool noverbose);
    virtual void mc_sample_averaging(PnlVect *theta, double &price, double &var, b
    virtual void mc_sample_averaging_delta(PnlVect *theta, double &price, double &

protected:
    size_t numberOfSamplesSAA;
    using MonteCarloCrude<Mode>::CurrentMode;
    using MonteCarloCrude<Mode>::finalize;
    using MonteCarloCrude<Mode>::finalize_delta;
    using MonteCarloCrude<Mode>::prepare_delta;
    using MonteCarloBase::rng;
    using MonteCarloBase::numberOfSamples;
    using MonteCarloBase::step;
    using MonteCarloBase::fdStep;
    using MonteCarloBase::expectation_order_n;
};

/**
 * Class used to implement numerical routines: Monte Carlo
 * and all the different stochastic algorithms. The kind of
 * importance sampling FULL or REDUCED is parametrized by
 * the template \ <Mode\ >.
 */
template<class Mode> class MonteCarloJumpHelper: public MonteCarloHelper<Mode>
{
public:
    MonteCarloJumpHelper();
    MonteCarloJumpHelper(PnlRng *rng, const Param &ParamTab);

```

```

~MonteCarloJumpHelper();

void expectation_order_n_poisson(PnlVect *const *sample_poisson, const PnlVect *
void expectation_order_n_gaussian_poisson(PnlVect *const *sample_G, const PnlVect *
virtual void sample_averaging_newton_poisson(PnlVect *mu, bool noverbose);
virtual void mc_sample_averaging_poisson(PnlVect *mu, double &price, double &variance);
virtual void sample_averaging_newton_gaussian_poisson(PnlVect *theta, PnlVect *mu);
virtual void mc_sample_averaging_gaussian_poisson(PnlVect *theta, PnlVect *mu, double &price, double &variance);

protected:
    using MonteCarloHelper<Mode>::numberOfSamplesSAA;
    using MonteCarloCrude<Mode>::CurrentMode;
    using MonteCarloCrude<Mode>::finalize;
    using MonteCarloCrude<Mode>::finalize_delta;
    using MonteCarloCrude<Mode>::prepare_delta;
    using MonteCarloBase::rng;
    using MonteCarloBase::numberOfSamples;
    using MonteCarloBase::step;
    using MonteCarloBase::fdStep;
    using MonteCarloBase::expectation_order_n;
};

// Import source code because all is templated
#include "MonteCarlo.cpp"

#endif

```