Help

```
#ifndef _MONTECARLOMODE_HPP
#define _MONTECARLOMODE_HPP


#include "
href../../../../common/math/ImportanceSampling_jl/src/Model_h_src.pdfmath/Import
#include "
href../../../../common/math/ImportanceSampling_jl/src/Option_h_src.pdfmath/Impor
#include "pnl/pnl_matrix.h"
#include "pnl/pnl_vector.h"
#include "pnl/pnl_random.h"

/// Full Monte Carlo mode helper
/// Overload path sampling methods from BaseModel to call the full versions
class MonteCarloModeFull
{
public:
  BaseModel *mod;
  BaseOption *opt;
  int IsSize;
  MonteCarloModeFull();
  MonteCarloModeFull(const Param &P);
  virtual ~MonteCarloModeFull();

  void print(bool verbose = false) const;
  inline PnlMat* getSamplePath() const { return mod->pathMatrix; }
  void path(PnlRng *rng) const;
  void path(PnlRng *rng, const PnlVect *drift) const;
  void path() const;
  void path(const PnlVect *drift) const;
  virtual double payoff() const;

  /**
   * Compute Grisanov's weight
   * @param G matrix of the Gaussian r.v. used to build the Brownian
   * @param drift theta full vector (one drift per time step)
   * @param isplus gives the sign within the exponential
   *    if isplus == true, return exp(-theta.G+|theta|^2/2)
   *    if isplus == false, return exp(-theta.G-|theta|^2/2)
```

```
   */
  double gaussianWeight(const PnlMat *G, const PnlVect *drift, bool isplus = fal

  /**
   * Compute the weight of the gradient in the Girsanov transformation
   *  -G exp(-2 * theta.G - |theta|^2)
   *
   *
   * @param[out] res holds the result on output
   * @param G the matrix of the Gaussian increment (normalized)
   * @param x the drift vector (one drift per time step)
   */
  void gaussianGradWeight(PnlVect *res, const PnlMat *G, const PnlVect *x) const

  /**
   * Return the random vector involved in Girsanov's weight
   *
   * @param G Matrix of renormalized brownian increments
   *
   * @return G as a vector
   */
  PnlVect* getGaussianIsVariable(const PnlMat *G) const;

  inline double getConstant() const { return 1.; }
};

class MonteCarloModeFullMultiLevel: public MonteCarloModeFull
{
private:
  int numberOfStepsLevel1;
public:
  BaseModel *coarseModel;
  BaseOption *coarseOption;
  MonteCarloModeFullMultiLevel();
  MonteCarloModeFullMultiLevel(const Param &P);
  virtual ~MonteCarloModeFullMultiLevel();
  virtual double payoff() const;
};

class MonteCarloJumpModeFull: public MonteCarloModeFull
{
```

```
public:
  JumpModel *dynamic_model;
  MonteCarloJumpModeFull();
  MonteCarloJumpModeFull(BaseModel *m, BaseOption *o);
  MonteCarloJumpModeFull(const Param &P);
  ~MonteCarloJumpModeFull();

  void pathMuTheta(PnlRng *rng, const PnlVect *drift, const PnlVect *mu);
  void pathMu(PnlRng *rng, const PnlVect *mu);

  /**
   * Return the intensities of the poissonMat increments as a vector
   *
   * @return
   */
  inline PnlVect* getJumpIntensity() const
    {
      return dynamic_model->lambdaFull;
    }

  /**
   * Get the poissonMat increments as a PnlMat
   *
   * @return
   */
  inline PnlMat* getPoissonIncr() const
    {
      return dynamic_model->poissonMat;
    }

  /**
   * computes prod(exp(mu_i-lambda_i)(lambda_i/mu_i)^number_of_jumps)
   * @param lambda initial parameter of poissonMat
   * @param mu new parameter of poissonMat
   * @param myPoiss a matrix such that myPoiss->mn == mu->size
   */
  double poissonWeight(const PnlVect *lambda, const PnlVect *mu, const PnlMat *m

  /**
   * Return the random vector involved in the poissonMat transformation
   *
```

```
   * @param P Matrix of poissonMat increments
   *
   * @return P as a vector
   */
  PnlVect* getPoissonIsVariable(const PnlMat *P) const;
};

/// Reduced Monte Carlo mode helper
/// Overload path sampling methods from BaseModel to call the reduced versions
class MonteCarloModeReduced
{
public:
  BaseModel *mod;
  BaseOption *opt;
  int IsSize;
  MonteCarloModeReduced();
  MonteCarloModeReduced(const Param &P);
  virtual ~MonteCarloModeReduced();
  void print(bool verbose = false) const;

  inline PnlMat* getSamplePath() const { return mod->pathMatrix; }

  void path(PnlRng *rng) const;
  void path(PnlRng *rng, const PnlVect *drift) const;
  void path() const;
  void path(const PnlVect *drift) const;
  virtual double payoff() const;

  /**
   * Computes  martingale weight corresponding the Cameron-Martin theorem
   *
   * @param G matrix of the Gaussian r.v. used to build the Brownian
   * @param x is the drift vector
   * @param isplus gives the sign within the exponential
   *    if isplus == true, return exp(-theta.G+|theta|^2/2)
   *    if isplus == false, return exp(-theta.G-|theta|^2/2)
   *
   * @return exp( -x . g sqrt(maturity) - x^2 * maturity / 2)
   */
  double gaussianWeight(const PnlMat *G, const PnlVect *x, bool isplus = false);
```

```
   /**
   * Computes the gradient of the variance in the case of the Cameron Martin
   * theorem
   *
   * @param[out] res holds the gradient of the Girsanov payoff on exit
   * @param G holds the increments of the Brownian path
   * @param x is the drift vector
   *
   * @return -g / sqrt(maturity) *  exp (-2 g . x * sqrt(maturity) - x^2 * matur
   */
  void gaussianGradWeight(PnlVect *res, const PnlMat *G, const PnlVect *x);

 /**
   * Return the random vector involved in Girsanov's weight
   *
   * @param G Matrix of renormalized brownian increments
   *
   * @return B_T
   */
  PnlVect* getGaussianIsVariable(const PnlMat *G) const;

  inline double getConstant() const { return mod->maturity; }

protected:
  PnlVect *BT;
 };

class MonteCarloModeReducedMultiLevel: public MonteCarloModeReduced
{
private:
  int numberOfStepsLevel1;
public:
  BaseModel *coarseModel;
  BaseOption *coarseOption;
  MonteCarloModeReducedMultiLevel();
  MonteCarloModeReducedMultiLevel(const Param &P);
  virtual ~MonteCarloModeReducedMultiLevel();
  virtual double payoff() const;
};

class MonteCarloJumpModeReduced: public MonteCarloModeReduced
```

```
{
protected:
  PnlVect *NT;

public:
  JumpModel *dynamic_model;

  MonteCarloJumpModeReduced();
  MonteCarloJumpModeReduced(const Param &P);
  ~MonteCarloJumpModeReduced();

  void pathMuTheta(PnlRng *rng, const PnlVect *drift, const PnlVect *mu) const;

  void pathMu(PnlRng *rng, const PnlVect *mu) const;

  /**
   * computes prod(exp(mu_i-lambda_i)(lambda_i/mu_i)^(NT_i))
   * @param lambda initial parameter of poissonMat
   * @param mu new parameter of poissonMat
   * @param myPoiss a matrix such that myPoiss->mn == mu->size
   */
  double poissonWeight(const PnlVect *lambda, const PnlVect *mu, const PnlMat *m

  /**
   * Return the intensities of the poissonMat increments as a vector
   *
   * @return
   */
  inline PnlVect* getJumpIntensity() const
    {
      return dynamic_model->lambda;
    }

  /**
   * Get the poissonMat increments as a PnlMat
   *
   * @return
   */
  inline PnlMat* getPoissonIncr() const
    {
      return dynamic_model->poissonMat;
```

```
    }

 /**
   * Return the random vector involved in the poissonMat transformation
   *
   * @param P Matrix of poissonMat increments
   *
   * @return P as a vector
   */
  PnlVect* getPoissonIsVariable(const PnlMat *P) const;

};

#endif /* _MONTECARLOMODE_HPP */
```