Help

```
#include <iostream>
#include <cstdlib>
#include <cstring>

#include "
href../../../../common/math/ImportanceSampling_jl/src/math_def_h_src.pdfmath/Imp
#include "
href../../../../common/math/ImportanceSampling_jl/src/MonteCarlo_h_src.pdfmath/I
#include "
href../../../../common/math/ImportanceSampling_jl/src/MonteCarloMode_h_src.pdfma
#include "
href../../../../common/math/ImportanceSampling_jl/src/MertonModel_h_src.pdfmath/

static void _set_diag_vect(PnlMat *M, const PnlVect *x)
{
  int i;
  for (i = 0 ; i < M->m ; i++)
    {
      MLET(M, i, i) = GET(x, i);
    }
}

/**
 * Compute the next iterate of the Newton algorithm making sure that all
 * components remain positive
 *
 * @param mu current iterate
 * @param d descent vector
 */
static void projection_step(PnlVect *mu, PnlVect *d)
{
  for (int i = 0 ; i < mu->size ; i++)
    {
      double tmp = GET(mu, i) - GET(d, i);
      if (tmp < 0)
        {
          LET(mu, i) /= 2.;
        }
      else
```

```
        {
          LET(mu, i) = tmp;
        }
    }
}

template <class Mode>
void MonteCarloCrude<Mode>::prepare_delta(PnlVect *delta, PnlVect *std_devdelta)
{
  pnl_vect_resize(delta, CurrentMode.mod->size);
  pnl_vect_resize(std_devdelta, CurrentMode.mod->size);
  pnl_vect_set_zero(delta);
  pnl_vect_set_zero(std_devdelta);
}

template <class Mode>
void MonteCarloCrude<Mode>::finalize_delta(PnlVect *delta, PnlVect *std_devdelta
{
  pnl_vect_mult_double(delta, std::exp(-CurrentMode.mod->interest * CurrentMode.
  pnl_vect_div_vect_term(delta, CurrentMode.mod->init);
  pnl_vect_mult_double(std_devdelta, std::exp(-2 * CurrentMode.mod->interest * C
  pnl_vect_div_vect_term(std_devdelta, CurrentMode.mod->init);
  pnl_vect_div_vect_term(std_devdelta, CurrentMode.mod->init);
  PnlVect *sqdelta = pnl_vect_copy(delta);
  pnl_vect_mult_vect_term(sqdelta, delta);
  pnl_vect_minus_vect(std_devdelta, sqdelta);
  pnl_vect_free(&sqdelta);
  pnl_vect_div_double(std_devdelta, double(numberOfSamples));
  pnl_vect_map_inplace(std_devdelta, std::sqrt);
}

template <class Mode>
void MonteCarloCrude<Mode>::finalize(double &price, double &std_dev, size_t Nsam
{
  price = price * std::exp(-CurrentMode.mod->interest * CurrentMode.opt->maturit
  std_dev = std::exp(-2 * CurrentMode.mod->interest * CurrentMode.opt->maturity)
  std_dev = std::sqrt(std_dev / double(Nsamples));
}

MonteCarloBase::MonteCarloBase()
{
```

```
  rng = NULL;
  numberOfSamples = 0;
  fdStep = 0.;
  step = 0;
}

/**
 * Constructor
 * @param p_rng a PnlRng
 * @param P a hash map
 */
MonteCarloBase::MonteCarloBase(PnlRng *p_rng, const Param &P)
{
  rng = p_rng;
  // step size for Robbins Monro (constant step)
  fdStep = 0.1;
  P.extract("sample number", numberOfSamples);
  P.extract("FD step", fdStep, true);
  P.extract("step size", step, true);
}

MonteCarloBase::~MonteCarloBase() { }


//
// Crude Monte-Carlo
//

template <class Mode>
MonteCarloCrude<Mode>::MonteCarloCrude() :
  MonteCarloBase(), CurrentMode()
{ }

/**
 * Constructor
 * @param rng a PnlRng
 * @param P a hash map
 */
template <class Mode>
MonteCarloCrude<Mode>::MonteCarloCrude(PnlRng *rng, const Param &P) :
  MonteCarloBase(rng, P), CurrentMode(P)
```

```
{ }

template <class Mode>
MonteCarloCrude<Mode>::~MonteCarloCrude() { }

template <class Mode>
void MonteCarloCrude<Mode>::print(bool verbose) const
{
  std::cout << std::endl;
  std::cout << "************************************" << std::endl;
  CurrentMode.print(verbose);
  std::cout << " Number of Samples : " << numberOfSamples << std::endl;
  std::cout << " step : " << step << std::endl;
  std::cout << "************************************" << std::endl << std::endl

}

/**
 * Computes the price by a crude Monte--Carlo estimator
 *
 * @param[out] prix price
 * @param[out] std_dev std_dev of the discounted payoff
 */
template <class Mode> void
MonteCarloCrude<Mode>::Price(double &prix, double &std_dev)
{
  double payoffVector;
  prix = 0.0;
  std_dev = 0.0;
  for (size_t i = 0 ; i < numberOfSamples ; i++)
    {
      CurrentMode.path(rng);
      payoffVector = CurrentMode.payoff();
      prix += payoffVector;
      std_dev += payoffVector * payoffVector;
    }
  finalize(prix, std_dev, numberOfSamples);
}

/**
 * Computes the price and delta
```

```
 *
 * @param[out] prix price
 * @param[out] std_devprix std_dev of the discounted payoff
 * @param[out] delta delta of the option
 * @param[out] std_devdelta std_dev of the FD estimator
 */
template <class Mode> void
MonteCarloCrude<Mode>::PriceDelta(double &prix, double &std_devprix, PnlVect *de
{
  double payoffVector;
  prix = 0.0;
  std_devprix = 0.0;

  /* resize delta ... */
  prepare_delta(delta, std_devdelta);
  for (size_t i = 0 ; i < numberOfSamples ; i++)
    {
      CurrentMode.path(rng);
      payoffVector = CurrentMode.payoff();
      prix += payoffVector;
      std_devprix += payoffVector * payoffVector;

      for (int d = 0 ; d < CurrentMode.mod->size ; d++)
        {
          double tmp_delta = 0.;
          CurrentMode.mod->shiftPath(d, fdStep);
          tmp_delta += CurrentMode.payoff();
          CurrentMode.mod->shiftPath(d, -2 * fdStep / (1 + fdStep));
          tmp_delta -= CurrentMode.payoff();
          CurrentMode.mod->unshiftPath(d, -fdStep);
          LET(delta, d) += tmp_delta;
          LET(std_devdelta, d) += tmp_delta * tmp_delta;
        }
    }

  finalize(prix, std_devprix, numberOfSamples);
  finalize_delta(delta, std_devdelta);
}

/**
 * Computes the price and stores the evolution of the estimator to a file
```

```
 *
 * @param[in] price_file file to which the evolution of the estimator is
 * saved
 * @param[out] prix price
 * @param[out] std_dev std_dev of the discounted payoff
 */
template <class Mode> void
MonteCarloCrude<Mode>::Price_evol(const char *price_file, double &prix, double &
{
  double payoffVector;
  FILE *DATA_price;
  DATA_price = fopen(price_file, "w");
  if (DATA_price == NULL)
    {
      perror("pb open file in MonteCarlo<Mode>::Price_evol");
      exit(1);
    }

  prix = 0.0;
  std_dev = 0.0;
  for (size_t i = 0 ; i < numberOfSamples ; i++)
    {
      CurrentMode.path(rng);
      payoffVector = CurrentMode.payoff();
      prix += payoffVector;
      std_dev += payoffVector * payoffVector;
      fprintf(DATA_price, "%f\ n", prix / (double)(i + 1));
    }

  finalize(prix, std_dev, numberOfSamples);
  fclose(DATA_price);
}

//
// Standard Robbins-Monro (adaptive and not)
//
template <class Mode>
MonteCarloHelper<Mode>::MonteCarloHelper(): MonteCarloCrude<Mode>(), numberOfSam

template <class Mode>
MonteCarloHelper<Mode>::MonteCarloHelper(PnlRng *rng, const Param &ParamTab) :
```

```
  MonteCarloCrude<Mode>(rng, ParamTab)
{
  numberOfSamplesSAA = numberOfSamples;
  ParamTab.extract("sample number SAA", numberOfSamplesSAA, true);
  if (numberOfSamplesSAA > NUMBER_OF_SAMPLES_SAA_MAX) numberOfSamplesSAA = NUMBE
}

template <class Mode>
MonteCarloHelper<Mode>::~MonteCarloHelper() { }

template <class Mode> void MonteCarloHelper<Mode>::print(bool verbose) const
{
  std::cout << std::endl;
  std::cout << "************************************" << std::endl;
  CurrentMode.print(verbose);
  std::cout << " Number of Samples : " << numberOfSamples << std::endl;
  std::cout << " Number of Samples SAA : " << numberOfSamplesSAA << std::endl;
  std::cout << " FD step : " << fdStep << std::endl;
  std::cout << " step : " << step << std::endl;
  std::cout << "************************************" << std::endl << std::endl
}

/**
 * Computes the price combined with a IS estimator based on a decreasing gain
 * stochastic approximation. This is a non adaptive approach
 *
 * @param[in] file_name file to which the evolution of the estimator is
 * saved
 * @param[out] price price
 * @param[out] std_dev std_dev of the discounted payoff
 *
 * @param[out] theta drift vector
 * @param[out] number_projection number of projection during the algorithm
 */
template <class Mode> void
MonteCarloHelper<Mode>::RM_routine(const char *file_name, PnlVect *theta, double
{
  FILE *OUTPUT = (FILE *) NULL;
  number_projection = 0;
  if (file_name != (char *) NULL)
    {
```

```
    OUTPUT = fopen(file_name, "w");
  }

price = 0;
std_dev = 0.0;
double price_tmp;
double payoff_val;
PnlVect *dvar_girsanov_tmp = pnl_vect_new();
pnl_vect_resize(theta, CurrentMode.IsSize);
pnl_vect_set_zero(theta);

for (size_t i = 0 ; i < numberOfSamples / 2 ; i++)
  {
    CurrentMode.path(rng, theta);
    payoff_val = CurrentMode.payoff();
    CurrentMode.gaussianGradWeight(dvar_girsanov_tmp, CurrentMode.mod->Gincr,
    pnl_vect_mult_double(dvar_girsanov_tmp, payoff_val * payoff_val * step / (
    pnl_vect_minus_vect(theta, dvar_girsanov_tmp);
    if (BaseOption::projection(theta, number_projection))
      {
        ++number_projection;
        pnl_vect_set_zero(theta);
      }

    if (OUTPUT != (FILE *) NULL)
      {
        pnl_vect_fprint(OUTPUT, theta);
      }
  }

for (size_t i = 0 ; i < numberOfSamples / 2 ; i++)
  {
    CurrentMode.path(rng, theta);
    payoff_val = CurrentMode.payoff();
    price_tmp = payoff_val * CurrentMode.gaussianWeight(CurrentMode.mod->Gincr
    price += price_tmp;
    std_dev += price_tmp * price_tmp;
  }

finalize(price, std_dev, numberOfSamples / 2);
```

```
  if (OUTPUT != (FILE *) NULL) fclose(OUTPUT);
  pnl_vect_free(&dvar_girsanov_tmp);
}

/**
 * Computes the price coupled with a IS estimator based on a decreasing gain
 * stochastic approximation. This is a non adaptive approach
 * One drift vector is used per time step
 *
 * @param[in] file_name file to which the evolution of the estimator is
 * saved
 * @param[out] price price
 * @param[out] std_dev std_dev of the discounted payoff
 *
 * @param[out] theta drift vector
 * @param[out] number_projection number of projection during the algorithm
 */
template <class Mode> void
MonteCarloHelper<Mode>::RM_routine_coupled(const char *file_name, PnlVect *theta
{
  FILE *OUTPUT = (FILE *) NULL;
  number_projection = 0;
  if (file_name != (char *) NULL)
    {
      OUTPUT = fopen(file_name, "w");
    }

  price = 0;
  std_dev = 0.0;
  double price_tmp;
  double payoff_val;
  PnlVect *dvar_girsanov_tmp = pnl_vect_new();
  pnl_vect_resize(theta, CurrentMode.IsSize);
  pnl_vect_set_zero(theta);

  for (size_t i = 0 ; i < numberOfSamples ; i++)
    {
      CurrentMode.path(rng, theta);
      payoff_val = CurrentMode.payoff();

      /* update the Monte Carlo estimator */
```

```
      price_tmp = payoff_val * CurrentMode.gaussianWeight(CurrentMode.mod->Gincr
      price += price_tmp;
      std_dev += price_tmp * price_tmp;

      CurrentMode.gaussianGradWeight(dvar_girsanov_tmp, CurrentMode.mod->Gincr,
      pnl_vect_mult_double(dvar_girsanov_tmp, payoff_val * payoff_val * step / (
      pnl_vect_minus_vect(theta, dvar_girsanov_tmp);
      if (BaseOption::projection(theta, number_projection))
        {
          ++number_projection;
          pnl_vect_set_zero(theta);
        }

      if (OUTPUT != (FILE *) NULL)
        pnl_vect_fprint(OUTPUT, theta);
    }

  finalize(price, std_dev, numberOfSamples);
  if (OUTPUT != (FILE *) NULL) fclose(OUTPUT);

  pnl_vect_free(&dvar_girsanov_tmp);
}

//
// Robbins-Monro + averaging
//

/**
 * Computes the price combined with a IS estimator based on an averaging
 * stochastic approximation. This is a non adaptive approach
 *
 * @param[in] drift file to which the evolution of the drift is
 * saved
 * @param[in] prix file to which the evolution of the MC estimator is
 * saved
 * @param[out] price price
 * @param[out] std_dev std_dev of the discounted payoff
 *
 * @param[out] theta drift vector
 * @param[out] number_projection number of projection during the algorithm
 * @param[in] t width of the averaging window
```

```
 */
template <class Mode> void
MonteCarloHelper<Mode>::RM_routine_average(const char *drift, const char *prix,
{
  FILE *OUTPUT = (FILE *) NULL;
  FILE *PRICE = (FILE *) NULL;
  number_projection = 0;
  if (drift != (char *) NULL)
    {
      OUTPUT = fopen(drift, "w");
    }

  if (prix != (char *) NULL)
    {
      PRICE = fopen(prix, "w");
    }

  price = 0;
  std_dev = 0.0;
  double tmp;
  double payoff_val;
  PnlVect *dvar_girsanov_tmp = pnl_vect_new();
  PnlVect **theta_iterate = new PnlVect *[numberOfSamples / 2 + 1];
  theta_iterate[0] = pnl_vect_create_from_zero(CurrentMode.IsSize);
  int bottom_window = 0;
  double alpha = 0.95;
  double window = 1;
  pnl_vect_resize(theta, CurrentMode.IsSize);
  pnl_vect_set_zero(theta);

  for (size_t i = 0 ; i < numberOfSamples / 2 ; i++)
    {
      CurrentMode.path(rng, theta_iterate[i]);
      payoff_val = CurrentMode.payoff();
      theta_iterate[i + 1] = pnl_vect_copy(theta_iterate[i]);
      CurrentMode.gaussianGradWeight(dvar_girsanov_tmp, CurrentMode.mod->Gincr,
      pnl_vect_mult_double(dvar_girsanov_tmp, payoff_val * payoff_val * step / p
      pnl_vect_minus_vect(theta_iterate[i + 1], dvar_girsanov_tmp);
      if (BaseOption::projection(theta_iterate[i + 1], number_projection))
        {
          ++number_projection;
```

```
        pnl_vect_set_zero(theta_iterate[i + 1]);
      }

    /* window is full - start moving the window */
    if (i - bottom_window > t * pow(bottom_window + 1, alpha))
      {
        window = t * pow(bottom_window + 1, alpha);
        pnl_vect_mult_double(theta, pow(bottom_window, alpha) * t);
        pnl_vect_minus_vect(theta, theta_iterate[bottom_window]);
        pnl_vect_plus_vect(theta, theta_iterate[i + 1]);
        pnl_vect_div_double(theta, window);
        bottom_window++;
      }
    else
      pnl_vect_axpby(1. / window, theta_iterate[i + 1], 1., theta);


    if (OUTPUT != (FILE *) NULL)
      pnl_vect_fprint(OUTPUT, theta);
  }

for (size_t i = 1 ; i < numberOfSamples / 2 ; i++)
  {
    CurrentMode.path(rng, theta);
    payoff_val = CurrentMode.payoff();
    tmp = payoff_val * CurrentMode.gaussianWeight(CurrentMode.mod->Gincr, thet
    price += tmp;
    std_dev += tmp * tmp;

    if (PRICE != (FILE *) NULL)
      fprintf(PRICE, "%f\ n", price / i);
  }

finalize(price, std_dev, numberOfSamples / 2);
for (size_t i = 0 ; i < numberOfSamples / 2 + 1 ; i++)
  pnl_vect_free(&theta_iterate[i]);

delete [] theta_iterate;
if (OUTPUT != (FILE *) NULL) fclose(OUTPUT);
if (PRICE != (FILE *) NULL) fclose(PRICE);
pnl_vect_free(&dvar_girsanov_tmp);
```

```
}

/**
 * Computes the price coupled with a IS estimator based on an averaging
 * stochastic approximation. This is an adaptive approach
 *
 * @param[in] drift file to which the evolution of the drift is
 * saved
 * @param[in] prix file to which the evolution of the MC estimator is
 * saved
 * @param[out] price price
 * @param[out] std_dev std_dev of the discounted payoff
 *
 * @param[out] theta drift vector
 * @param[out] number_projection number of projection during the algorithm
 * @param[in] t width of the averaging window
 */
template <class Mode> void
MonteCarloHelper<Mode>::RM_routine_average_coupled(const char *drift, const char
{
  FILE *OUTPUT = (FILE *) NULL;
  FILE *PRICE = (FILE *) NULL;
  number_projection = 0;
  if (drift != (char *) NULL)
    {
      OUTPUT = fopen(drift, "w");
    }

  if (prix != (char *) NULL)
    {
      PRICE = fopen(prix, "w");
    }

  price = 0;
  std_dev = 0.0;
  double tmp;
  double payoff_val;
  PnlVect *dvar_girsanov_tmp = pnl_vect_new();
  PnlVect **theta_iterate = new PnlVect *[numberOfSamples + 1];
  theta_iterate[0] = pnl_vect_create_from_zero(CurrentMode.IsSize);
  int bottom_window = 0;
```

```
double alpha = 0.95;
double window = 1;
pnl_vect_resize(theta, CurrentMode.IsSize);
pnl_vect_set_zero(theta);

for (size_t i = 0 ; i < numberOfSamples ; i++)
  {
    CurrentMode.path(rng, theta);
    payoff_val = CurrentMode.payoff();

    /* updating the Monte Carlo estimator */
    tmp = payoff_val * CurrentMode.gaussianWeight(CurrentMode.mod->Gincr, thet
    price += tmp;
    std_dev += tmp * tmp;

    if (PRICE != (FILE *) NULL)
      fprintf(PRICE, "%f\ n", price / i);


    /* updating the averaging stochastic algorithm */
    CurrentMode.path(theta_iterate[i]);
    payoff_val = CurrentMode.payoff();
    theta_iterate[i + 1] = pnl_vect_copy(theta_iterate[i]);

    CurrentMode.gaussianGradWeight(dvar_girsanov_tmp, CurrentMode.mod->Gincr,
    pnl_vect_mult_double(dvar_girsanov_tmp, payoff_val * payoff_val * step / p
    pnl_vect_minus_vect(theta_iterate[i + 1], dvar_girsanov_tmp);
    if (BaseOption::projection(theta_iterate[i + 1], number_projection))
      {
        ++number_projection;
        pnl_vect_set_zero(theta_iterate[i + 1]);
      }

    /* window is full - start moving the window */
    if (i - bottom_window > t * pow(bottom_window + 1, alpha))
      {
        window = t * pow(bottom_window + 1, alpha);
        pnl_vect_mult_double(theta, pow(bottom_window, alpha) * t);
        pnl_vect_minus_vect(theta, theta_iterate[bottom_window]);
        pnl_vect_plus_vect(theta, theta_iterate[i + 1]);
        pnl_vect_div_double(theta, window);
```

```
          bottom_window++;
        }
      else
        pnl_vect_axpby(1. / window, theta_iterate[i + 1], 1., theta);

      if (OUTPUT != (FILE *) NULL)
        pnl_vect_fprint(OUTPUT, theta);
    }

  finalize(price, std_dev, numberOfSamples);
  for (size_t i = 0 ; i < numberOfSamples + 1 ; i++)
    pnl_vect_free(&theta_iterate[i]);

  delete [] theta_iterate;
  if (OUTPUT != (FILE *) NULL) fclose(OUTPUT);
  if (PRICE != (FILE *) NULL) fclose(PRICE);
  pnl_vect_free(&dvar_girsanov_tmp);
}

//
// Sample averaging approach
//

/**
 * Computes the different terms involved in the sample average minimization
 * in a way to reduce to computational cost
 *  (same drift vector for all time steps)
 *
 * \ f[
 * E(GG^\ prime \ phi(G)^2 e^{-\ theta \ cdot G} )
 * \ f]
 *
 * \ f[
 * E(G  \ phi(G)^2 e^{-\ theta \ cdot G})
 * \ f]
 *
 * \ f[
 * E( \ phi(G)^2 e^{-\ theta \ cdot G})
 * \ f]
 *
```

```
 * @param[in] g vectors of the final values of W_T
 * @param[in] theta current value of the drift vector
 * @param[in] payoffs vector of the payoffs computed on the paths built with
 * the vector sample_G
 * @param[out] expect_0
 * @param[out] expect_1
 * @param[out] expect_2
 */
template <class Mode> void
MonteCarloHelper<Mode>::expectation_order_n(PnlVect *const *g, const PnlVect *th
                                            double &expect_0, PnlVect *expect_1, P
{
  double tmp;
  expect_0 = 0.0;
  pnl_vect_resize(expect_1, theta->size);
  pnl_mat_resize(expect_2, theta->size, theta->size);
  pnl_vect_set_zero(expect_1);
  pnl_mat_set_zero(expect_2);

  for (size_t i = 0 ; i < numberOfSamplesSAA ; i++)
    {
      tmp = sq(GET(payoffs, i)) * std::exp(-(pnl_vect_scalar_prod(theta, g[i])))
      expect_0 += tmp;

      pnl_vect_axpby(tmp, g[i], 1., expect_1);   /* E1 += tmp  * g[i] */
      pnl_mat_dger(tmp, g[i], g[i], expect_2);   /* E2 += tmp  * g[i]' * g[i] */
    }
}

/**
 * Computes the optimal drift (one drift vector for all time steps) using the
 * random variables sample_G
 *
 * @param[in] theta drift vector (one for all time steps)
 * @param[in] noverbose if true nothing is printed about the intermediate steps
 * of the Newton procedure
 *
 * @return the optimal drift vector
 */
template <class Mode> void
MonteCarloHelper<Mode>::sample_averaging_newton(PnlVect *theta, bool noverbose)
```

```
{
  double expect_0, norm_gradv;
  PnlVect *payoffs = pnl_vect_create(numberOfSamplesSAA);
  PnlVect **G = new PnlVect *[numberOfSamplesSAA];
  PnlVect *expect_1 = pnl_vect_new();
  PnlVect *gradVector = pnl_vect_new();
  PnlMat *expect_2 = pnl_mat_new();
  PnlMat *hesVector = pnl_mat_create(CurrentMode.IsSize, CurrentMode.IsSize);
  double EPS = 0.00000001 * CurrentMode.IsSize;
  int k = 30;
  double constant_mode = CurrentMode.getConstant(); // maturity or 1 dependendin
  pnl_vect_resize(theta, CurrentMode.IsSize);
  pnl_vect_set_zero(theta);

  for (size_t i = 0 ; i < numberOfSamplesSAA ; i++)
    {
      CurrentMode.path(rng);
      LET(payoffs, i) = CurrentMode.payoff();
      G[i]  = CurrentMode.getGaussianIsVariable(CurrentMode.mod->Gincr);
    }


  for (int l = 0 ; l < k ; l++)
    {
      expectation_order_n(G, theta, payoffs, expect_0, expect_1, expect_2);
      pnl_vect_clone(gradVector, theta);
      pnl_vect_axpby(1. / (-constant_mode * expect_0), expect_1, 1., gradVector)

      /* hesVector = I + ( E2 E 0 + E1'E1) / (E0^2 maturity) */
      pnl_mat_div_double(expect_2, expect_0 * constant_mode);
      pnl_mat_set_id(hesVector);
      pnl_mat_plus_mat(hesVector, expect_2);
      pnl_mat_dger(-1. / (expect_0 * expect_0 * constant_mode), expect_1, expect

      norm_gradv = pnl_vect_norm_two(gradVector);
      pnl_mat_chol(hesVector);
      pnl_mat_chol_syslin_inplace(hesVector, gradVector);
      pnl_vect_axpby(-1., gradVector, 1., theta);   /* theta -= gradVector */

      if (!noverbose)
        std::cout << "iteration " << l << ", norm of the gradient : " <<  norm_g
```

```
      if (!noverbose && norm_gradv < EPS)
        {
          std::cout << "iteration : " <<  l << std::endl;
          break;
        }
    }
  for (size_t i = 0 ; i < numberOfSamplesSAA ; i++) pnl_vect_free(&(G[i]));
  delete [] G;

  pnl_vect_free(&payoffs);
  pnl_vect_free(&expect_1);
  pnl_vect_free(&gradVector);
  pnl_mat_free(&expect_2);
  pnl_mat_free(&hesVector);
}

/**
 * Computes the price using a sample average based IS estimator
 *
 * @param[out] theta computed drift
 * @param[out] price estimated price
 * @param[out] std_dev asymptotic std_dev of the estimator
 * @param[in] noverbose
 */
template <class Mode> void
MonteCarloHelper<Mode>::mc_sample_averaging(PnlVect *theta, double &price, doubl
{
  double tmp;
  price = 0.0;
  std_dev = 0.0;

  /* computation of the theta optimal */
  sample_averaging_newton(theta, noverbose);

  /* computation of MC with that value using independent samples */
  for (size_t i = 0 ; i < numberOfSamples ; i++)
    {
      CurrentMode.path(rng, theta);
      tmp = CurrentMode.payoff() * CurrentMode.gaussianWeight(CurrentMode.mod->G
      price += tmp;
```

```
      std_dev += tmp * tmp;
    }
  finalize(price, std_dev, numberOfSamples);
}

/**
 * Computes the price and deltas using a sample average based IS estimator (same
 * drift vector for all time steps)
 *
 * @param[out] theta computed drift
 * @param[out] price 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[in] noverbose
 */
template <class Mode> void
MonteCarloHelper<Mode>::mc_sample_averaging_delta(PnlVect *theta, double &price,
{
  double tmp;
  price = 0.0;
  std_dev = 0.0;

  /* computation of the theta optimal */
  sample_averaging_newton(theta, noverbose);

  /* computation of MC with that value using the same samples */
  /* resize delta ... */
  prepare_delta(delta, std_devdelta);

  for (size_t i = 0 ; i < numberOfSamples ; i++)
    {
      CurrentMode.path(rng, theta);
      tmp = CurrentMode.payoff() * CurrentMode.gaussianWeight(CurrentMode.mod->G
      price += tmp;
      std_dev += tmp * tmp;

      for (int d = 0 ; d < CurrentMode.mod->size ; d++)
        {
          double tmp_delta = 0.;
          double payoff_val;
```

```
          CurrentMode.mod->shiftPath(d, fdStep);
          payoff_val = CurrentMode.payoff();
          tmp_delta = payoff_val * CurrentMode.gaussianWeight(CurrentMode.mod->G
          CurrentMode.mod->unshiftPath(d, fdStep);

          CurrentMode.mod->shiftPath(d, -fdStep);
          CurrentMode.mod->unshiftPath(d, -fdStep);

          LET(delta, d) += tmp_delta;
          LET(std_devdelta, d) += tmp_delta * tmp_delta;
        }
    }

  finalize(price, std_dev, numberOfSamples);
  finalize_delta(delta, std_devdelta);
}


template <class Mode>
MonteCarloJumpHelper<Mode>::MonteCarloJumpHelper(): MonteCarloHelper<Mode>() { }

template <class Mode>
MonteCarloJumpHelper<Mode>::MonteCarloJumpHelper(PnlRng *rng, const Param &Param
  MonteCarloHelper<Mode>(rng, ParamTab) { }

template <class Mode>
MonteCarloJumpHelper<Mode>::~MonteCarloJumpHelper() { }

/**
 * Computes the different terms involved in the sample average minimization
 * in a way to reduce the computational cost.
 * (one intensity vector per time step)
 *
 * \ f[
 * E((N / mu)(N / mu)^\ prime \ phi^2 \ prod_i * \ frac{\ lambda_i}{\ mu_i}^{N_i
 * \ f]
 *
 * \ f[
 * E((N / mu)  \ phi^2 \ prod_i * \ frac{\ lambda_i}{\ mu_i}^{N_i} )
 * \ f]
 *
```

```
 * \ f[
 * E( \ phi^2 \ prod_i * \ frac{\ lambda_i}{\ mu_i}^{N_i} )
 * \ f]
 *
 * \ f[
 * E( diag(N / \ mu^2) \ phi^2 \ prod_i * \ frac{\ lambda_i}{\ mu_i}^{N_i} )
 * \ f]
 *
 * @param[in] sample_poiss an array of vectors, each component corresponds to th
 * r.v. needed to build one path of the model
 * @param[in] mu current value of the intensity vector
 * @param[in] lambda original value of the intensity vector
 * @param[in] payoffs vector of the payoffs computed on the paths built
 * @param[out] expect_0
 * @param[out] expect_1
 * @param[out] expect_2
 * @param[out] diag_hess
 */
template <class Mode> void
MonteCarloJumpHelper<Mode>::expectation_order_n_poisson(PnlVect *const *sample_p
                                                        const PnlVect *lamb
                                                        PnlMat *expect_2, P
{
  PnlVect *N_over_mu, *N_over_mu2;
  expect_0 = 0.0;
  pnl_vect_resize(expect_1, mu->size);
  pnl_vect_resize(diag_hess, mu->size);
  pnl_mat_resize(expect_2, mu->size, mu->size);
  pnl_vect_set_zero(expect_1);
  pnl_vect_set_zero(diag_hess);
  pnl_mat_set_zero(expect_2);
  N_over_mu = pnl_vect_create(mu->size);
  N_over_mu2 = pnl_vect_create(mu->size);

  for (size_t k = 0 ; k < numberOfSamplesSAA ; k++)
    {
      double prod = 1.0;
      for (int i = 0 ; i < mu->size ; i++)
        {
          const double lambda_i = GET(lambda, i);
          const double mu_i = GET(mu, i);
```

```
          const double N_i = GET(sample_poiss[k], i);
          const double Ni_mui = N_i / mu_i;
          LET(N_over_mu2, i) = Ni_mui / mu_i;
          LET(N_over_mu, i) = Ni_mui;
          prod *= pnl_pow_i((lambda_i / mu_i), (int) N_i);
        }

      double tmp = sq(GET(payoffs, k)) * prod;
      expect_0 += tmp;
      pnl_vect_axpby(tmp, N_over_mu, 1., expect_1);   // E_1 += N/mu * tmp;
      pnl_vect_axpby(tmp, N_over_mu2, 1., diag_hess);   // diag_hess += N/m^2u *
      pnl_mat_dger(tmp, N_over_mu, N_over_mu, expect_2);   // E_2 += tmp * N/mu
    }
  pnl_vect_free(&N_over_mu);
  pnl_vect_free(&N_over_mu2);
}

/**
 * Computes the different terms involved in the sample average minimization
 * in a way to reduce the computational cost.
 * (same intensity vector for all time steps)
 *
 * Let Z = (G, N/mu)
 * \ f[
 * E(ZZ' \ phi^2 \ prod_i * \ frac{\ lambda_i}{\ mu_i}^{N_i} e^{-\ theta \ cdot
 * \ f]
 *
 * \ f[
 * E(Z  \ phi^2 \ prod_i * \ frac{\ lambda_i}{\ mu_i}^{N_i} e^{-\ theta \ cdot G
 * \ f]
 *
 * \ f[
 * E( \ phi^2 \ prod_i * \ frac{\ lambda_i}{\ mu_i}^{N_i} e^{-\ theta \ cdot G})
 * \ f]
 *
 * \ f[
 * E( (1 diag(N / \ mu^2)) \ phi^2 \ prod_i * \ frac{\ lambda_i}{\ mu_i}^{N_i} e
 * \ f]
 *
 * @param[in] G an array of vectors, each component corresponds to the
 * r.v. needed to build one path of the Brownian motion
```

```
 * @param[in] theta drift for the gaussian part
 * @param[in] poiss an array of vectors, each component corresponds to the
 * r.v. needed to build one path of the jumps
 * @param[in] lambda original value of the jump intensity
 * @param[in] mu current value for the IS jump intensity
 * @param[in] payoffs vector of the payoffs computed on the paths built
 * @param[out] expect_0
 * @param[out] expect_1
 * @param[out] expect_2
 * @param[out] diag_hess
 */
template <class Mode> void
MonteCarloJumpHelper<Mode>::expectation_order_n_gaussian_poisson(PnlVect *const
                                                                 PnlVect *c
                                                                 double &ex
{
  int size = mu->size + theta->size;
  expect_0 = 0.0;
  pnl_vect_resize(expect_1, size);
  pnl_vect_resize(diag_hess, size);
  pnl_mat_resize(expect_2, size, size);
  pnl_vect_set_zero(expect_1);
  pnl_vect_set_zero(diag_hess);
  pnl_mat_set_zero(expect_2);
  PnlVect *Z = pnl_vect_create(size);
  PnlVect Ztheta = pnl_vect_wrap_subvect(Z, 0, theta->size);
  PnlVect Zmu = pnl_vect_wrap_subvect(Z, theta->size, mu->size);


  for (size_t k = 0 ; k < numberOfSamplesSAA ; k++)
    {
      double prod = 1.0;
      for (int i = 0 ; i < mu->size ; i++)
        {
          const double lambda_i = GET(lambda, i);
          const double mu_i = GET(mu, i);
          const double N_i = GET(poiss[k], i);
          prod *= pow((lambda_i / mu_i), N_i);
        }

      double tmp = sq(GET(payoffs, k)) * prod * exp(-pnl_vect_scalar_prod(theta,
```

```
      expect_0 += tmp;
      pnl_vect_clone(&Zmu, poiss[k]);
      pnl_vect_clone(&Ztheta, G[k]);
      pnl_vect_div_vect_term(&Zmu, mu);
      pnl_vect_axpby(tmp, Z, 1., expect_1);   // E_1 += (G N/mu) * tmp;
      pnl_mat_dger(tmp, Z, Z, expect_2);   // E_2 += tmp * (G N/mu) * (G N/mu)'
      for (int i = 0 ; i < mu->size ; i++)
        {
          const double mu_i = GET(mu, i);
          const double N_i = GET(poiss[k], i);
          LET(diag_hess, i + theta->size) += N_i / (mu_i * mu_i) * tmp;
        }
    }
  for (int i = 0 ; i < theta->size ; i++)
    {
      LET(diag_hess, i) = expect_0;
    }
  pnl_vect_free(&Z);
}

/**
 * Compute the Newton method for the JumpModel BaseModel (poissonMat reduction)
 *
 * @param[out] mu new parameter for the poissonMat distribution
 * @param[in] noverbose
 */
template <class Mode> void
MonteCarloJumpHelper<Mode>::sample_averaging_newton_poisson(PnlVect *mu,  bool n
{
  PnlVect *expect_1 = pnl_vect_new();
  PnlVect *diag_hess = pnl_vect_new();
  PnlMat *expect_2 = pnl_mat_new();
  PnlVect *payoffs = pnl_vect_create(this->numberOfSamplesSAA);
  PnlVect **poiss_final = new PnlVect *[this->numberOfSamplesSAA];
  PnlVect *grad = pnl_vect_create(mu->size);
  PnlMat *hes = pnl_mat_create(mu->size, mu->size);
  double EPS = 0.00001 * CurrentMode.mod->brownianSize;
  int nb_iter = 30;

  for (size_t i = 0 ; i < numberOfSamplesSAA ; i++)
    {
```

```
    CurrentMode.path(this->rng);
    poiss_final[i] = CurrentMode.getPoissonIsVariable(CurrentMode.dynamic_mode
    LET(payoffs, i) = CurrentMode.payoff();
  }

for (int l = 0 ; l < nb_iter ; l++)
  {
    double norm_grad, expect_0;

    expectation_order_n_poisson(poiss_final, mu, CurrentMode.getJumpIntensity(

    pnl_mat_set_zero(hes);
    // If CurrentMode=Full, set to 1. If CurrentMode=Reduced, set to mod->matu
    pnl_vect_set_double(grad, CurrentMode.getConstant());
    pnl_vect_axpby(-1. / expect_0, expect_1, 1., grad);   // grad = -E_1 / E_0

    /* hes = diag_hess / E_0 + E_2 / E_0 - E_1E_1' / (E_0^2) */
    pnl_mat_div_double(expect_2, expect_0);
    pnl_vect_div_double(diag_hess, expect_0);
    _set_diag_vect(hes, diag_hess);
    pnl_mat_plus_mat(hes, expect_2);
    pnl_mat_dger(-1. / (expect_0 * expect_0), expect_1, expect_1, hes);

    norm_grad = pnl_vect_norm_two(grad);
    // Newton's method - resolution of the system
    pnl_mat_chol(hes);
    pnl_mat_chol_syslin_inplace(hes, grad);
    projection_step(mu, grad);
    if (!noverbose)
      printf("iteration %d, norm of the gradient : %f\ n", l, norm_grad);
    if (norm_grad < EPS) break;
  }

for (size_t i = 0 ; i < numberOfSamplesSAA ; i++) pnl_vect_free(&(poiss_final[
delete [] poiss_final;
pnl_vect_free(&grad);
pnl_vect_free(&payoffs);
pnl_mat_free(&hes);
pnl_vect_free(&expect_1);
pnl_vect_free(&diag_hess);
pnl_mat_free(&expect_2);
```

```
}

/**
 * Reduction of std_dev for JumpModel BaseModel (poissonMat Reduction)
 *
 * @param[out] prix price
 * @param[out] std_dev std_dev
 * @param[out] mu new parameter of the poissonMat distribution
 * @param[in] noverbose
 */
template <class Mode> void
MonteCarloJumpHelper<Mode>::mc_sample_averaging_poisson(PnlVect *mu, double &pri
{
  pnl_vect_clone(mu, CurrentMode.getJumpIntensity());

  /* Computation of the optimal mu*/
  sample_averaging_newton_poisson(mu, noverbose);

  prix = 0.;
  std_dev = 0.0;
  for (size_t i = 0 ; i < numberOfSamples ; i++)
    {
      CurrentMode.pathMu(rng, mu);
      double payoff_val = CurrentMode.payoff();
      double weight = CurrentMode.poissonWeight(CurrentMode.getJumpIntensity(),
      payoff_val *= weight;
      prix += payoff_val;
      std_dev += payoff_val * payoff_val;
    }

  finalize(prix, std_dev, numberOfSamples);
}

/**
 * Computes the Newton method for the reduction Gaussian and poissonMat
 *
 * @param[out] theta value of the drift vector
 * @param[out] mu new parameter of the poissonMat distribution
 * @param[in] noverbose
 */
template <class Mode> void
```

```
MonteCarloJumpHelper<Mode>::sample_averaging_newton_gaussian_poisson(PnlVect *th
{
  int nb_param = theta->size + mu->size;
  PnlVect *expect_1 = pnl_vect_new();
  PnlVect *diag_hess = pnl_vect_new();
  PnlMat *expect_2 = pnl_mat_new();
  PnlVect **g_final = new PnlVect* [this->numberOfSamplesSAA];
  PnlVect **poiss_final = new PnlVect* [this->numberOfSamplesSAA];
  PnlVect *payoffs = pnl_vect_create(this->numberOfSamplesSAA);
  PnlVect *grad = pnl_vect_create_from_zero(nb_param);
  PnlVect grad_theta = pnl_vect_wrap_subvect(grad, 0 , theta->size);
  PnlVect grad_mu = pnl_vect_wrap_subvect(grad, theta->size, mu->size);
  PnlMat *hes = pnl_mat_create(nb_param, nb_param);
  double EPS = 0.00001 * nb_param;
  int nb_iter = 30;


  for (size_t i = 0 ; i < this->numberOfSamplesSAA ; i++)
    {
      CurrentMode.path(this->rng);
      LET(payoffs, i) = this->CurrentMode.payoff();
      g_final[i]  = CurrentMode.getGaussianIsVariable(CurrentMode.mod->Gincr);
      poiss_final[i] = CurrentMode.getPoissonIsVariable(CurrentMode.dynamic_mode
    }

  for (int l = 0 ; l < nb_iter ; l++)
    {
      double norm_grad, expect_0;
      pnl_vect_set_zero(grad);
      pnl_mat_set_zero(hes);

      expectation_order_n_gaussian_poisson(g_final, theta, poiss_final, mu, Curr
      pnl_vect_clone(&grad_theta, theta);
      // If CurrentMode=Full, set to 1. If CurrentMode=Reduced, set to mod->matu
      pnl_vect_set_double(&grad_mu, CurrentMode.getConstant());
      pnl_vect_axpby(-1. / (expect_0), expect_1, 1., grad);   // grad = -E_1 / (

      /* hes = diag_hess / E_0 + E_2 / E_0 - E_1E_1' / (E_0^2) */
      pnl_mat_div_double(expect_2, expect_0);
      pnl_vect_div_double(diag_hess, expect_0);
      _set_diag_vect(hes, diag_hess);
```

```
      pnl_mat_plus_mat(hes, expect_2);
      pnl_mat_dger(-1. / (expect_0 * expect_0), expect_1, expect_1, hes);

      // Newton's method - resolution of the system
      norm_grad = pnl_vect_norm_two(grad);
      pnl_mat_chol(hes);
      pnl_mat_chol_syslin_inplace(hes, grad);
      pnl_vect_minus_vect(theta, &grad_theta);
      projection_step(mu, &grad_mu);
      if (!noverbose)
        {
          printf("iteration %d, norm of the gradient : %f\ n", l, norm_grad);
        }
      if (norm_grad < EPS) break;
    }

  for (size_t i = 0 ; i < this->numberOfSamplesSAA ; i++)
    {
      pnl_vect_free(&(poiss_final[i]));
      pnl_vect_free(&(g_final[i]));
    }
  delete [] poiss_final;
  delete [] g_final;
  pnl_vect_free(&grad);
  pnl_vect_free(&payoffs);
  pnl_vect_free(&expect_1);
  pnl_vect_free(&diag_hess);
  pnl_mat_free(&hes);
  pnl_mat_free(&expect_2);
}

/**
 * Reduction of std_dev for Gaussian and poissonMat parameters
 *
 * @param[out] theta value of the drift vector
 * @param[out] mu new parameter of the poissonMat distribution
 * @param[out] prix price
 * @param[out] std_dev std_dev for the price estimator
 * @param[in] noverbose
 */
template <class Mode> void
```

```
MonteCarloJumpHelper<Mode>::mc_sample_averaging_gaussian_poisson(PnlVect *theta,
{
  pnl_vect_resize(theta, CurrentMode.IsSize);
  pnl_vect_set_zero(theta);
  pnl_vect_clone(mu, CurrentMode.getJumpIntensity());

  /* Computation of the optimal (theta,mu) */
  sample_averaging_newton_gaussian_poisson(theta, mu, noverbose);

  /* Calculation of price + IC */
  prix = 0.0;
  std_dev = 0.0;
  for (size_t i = 0 ; i < numberOfSamples ; i++)
    {
      CurrentMode.pathMuTheta(rng, theta, mu);
      double payoff_val = CurrentMode.payoff();

      payoff_val = payoff_val *
       CurrentMode.poissonWeight(CurrentMode.getJumpIntensity(), mu, CurrentMode
       CurrentMode.gaussianWeight(CurrentMode.mod->Gincr, theta, false);
      prix += payoff_val;
      std_dev += payoff_val * payoff_val;
    }

  finalize(prix, std_dev, numberOfSamples);
}


MonteCarloBase* CreateMonteCarlo(PnlRng *rng, const Param &P, bool IsReduced)
{
  BaseModel *mod;
  MonteCarloBase *mc;
  if ((mod = instantiate_model(P)) == NULL)
    {
      printf("No valid model found. Aborting...\ n");
      abort();
    }

  JumpModel *j = dynamic_cast<JumpModel *>(mod);
  if (j != NULL)
    {
```

```
      if (IsReduced) mc = new MonteCarloJumpHelper<MonteCarloJumpModeReduced>(rn
      else  mc = new MonteCarloJumpHelper<MonteCarloJumpModeFull>(rng, P);
    }
  else
    {
      if (IsReduced) mc = new MonteCarloHelper<MonteCarloModeReduced>(rng, P);
      else  mc = new MonteCarloHelper<MonteCarloModeFull>(rng, P);
    }
  delete mod;
  return mc;
}
```