Help

```
#include "
href../../../../common/math/ImportanceSampling_jl/src/MonteCarloMode_h_src.pdfma

MonteCarloModeFull::MonteCarloModeFull(): mod(NULL), opt(NULL) { }

MonteCarloModeFull::MonteCarloModeFull(const Param &P)
{
  mod = instantiate_model(P);
  opt = instantiate_option(P);
  IsSize = mod->brownianSize * mod->nTimeSteps;
}

MonteCarloModeFull::~MonteCarloModeFull()
{
  if (mod != NULL) delete mod;
  if (opt != NULL) delete opt;
}

void MonteCarloModeFull::print(bool verbose) const
{
  std::cout << " Full mode" << std::endl;
  if (verbose)
    {
      mod->print();
      opt->print();
    }
  std::cout << "**** Monte Carlo Characteristics ****" << std::endl;
  std::cout << " Number of timesteps " << mod->nTimeSteps << std::endl;
}


void MonteCarloModeFull::path(PnlRng *rng) const { mod->path(rng); }

void MonteCarloModeFull::path(PnlRng *rng, const PnlVect *drift)  const { mod->p

void MonteCarloModeFull::path() const { mod->path(); }

void MonteCarloModeFull::path(const PnlVect *drift) const { mod->pathFull(drift)
```

```
double MonteCarloModeFull::payoff() const { return opt->payoff(getSamplePath());

double MonteCarloModeFull::gaussianWeight(const PnlMat *G, const PnlVect *drift,
{
  PnlVect g = pnl_vect_wrap_mat(G);
  double scalar = pnl_vect_scalar_prod(&g, drift);
  double norm = pnl_vect_scalar_prod(drift, drift);

  if (isplus) return exp(-scalar + norm / 2.);
  else return exp(-scalar - norm / 2);
}

void MonteCarloModeFull::gaussianGradWeight(PnlVect *res, const PnlMat *G, const
{
  PnlVect g = pnl_vect_wrap_mat(G);
  pnl_vect_clone(res, &g);
  pnl_vect_mult_double(res, - exp(-2 * (pnl_vect_scalar_prod(&g, x))
                                   - pnl_vect_scalar_prod(x, x)));

}

PnlVect* MonteCarloModeFull::getGaussianIsVariable(const PnlMat *G) const
{
  return pnl_vect_create_from_ptr(G->mn, G->array);
}

MonteCarloJumpModeFull::MonteCarloJumpModeFull() : MonteCarloModeFull() { }

MonteCarloJumpModeFull::MonteCarloJumpModeFull(const Param &P) :
  MonteCarloModeFull(P)
{
  dynamic_model = dynamic_cast<JumpModel *>(mod);
  if (dynamic_model == NULL)
    {
      std::cout << "Cannot convert BaseModel to JumpModel" << std::endl;
      abort();
    }
}

MonteCarloJumpModeFull::~MonteCarloJumpModeFull() { }
```

```
void MonteCarloJumpModeFull::pathMuTheta(PnlRng *rng, const PnlVect *drift, cons
{
  dynamic_model->pathMuThetaFull(rng, drift, mu);
}

void MonteCarloJumpModeFull::pathMu(PnlRng *rng, const PnlVect *mu)
{
  dynamic_model->pathMuFull(rng, mu);
}

double MonteCarloJumpModeFull::poissonWeight(const PnlVect *lambda, const PnlVec
{
  double prod = 1.0;
  PnlVect poiss = pnl_vect_wrap_mat(myPoiss);
  for (int i = 0 ; i < lambda->size ; i++)
    {
      double lambda_i = GET(lambda, i);
      double mu_i = GET(mu, i);
      double N_i = GET(&poiss, i);
      prod *= exp(mu_i - lambda_i) * pow((lambda_i / mu_i), N_i);
    }
  return prod;
}

PnlVect* MonteCarloJumpModeFull::getPoissonIsVariable(const PnlMat *Pois) const
{
  return pnl_vect_create_from_ptr(Pois->mn, Pois->array);
}

MonteCarloModeFullMultiLevel::MonteCarloModeFullMultiLevel() :
  MonteCarloModeFull(), coarseModel(NULL), coarseOption(NULL) { }

MonteCarloModeFullMultiLevel::MonteCarloModeFullMultiLevel(const Param &P) :
  MonteCarloModeFull(P)
{
  int fine_timesteps;
  Param Pcopy(P);
  P.extract("timestep number level one", numberOfStepsLevel1);
  P.extract("timestep number", fine_timesteps);
  Pcopy.set("timestep number", fine_timesteps / numberOfStepsLevel1);
  coarseModel = instantiate_model(Pcopy);
```

```
  coarseOption = instantiate_option(Pcopy);
}

MonteCarloModeFullMultiLevel::~MonteCarloModeFullMultiLevel()
{
  if (coarseModel != NULL) delete coarseModel;
  if (coarseOption != NULL) delete coarseOption;
}

/**
 * Compute the difference of the payoffs on the fine and coarse grids
 *
 * @return
 */
double MonteCarloModeFullMultiLevel::payoff() const
{
  double payoff_coarse, payoff_fine;
  payoff_fine = opt->payoff(mod->pathMatrix);
  // Agregate steps
  for (int j = 0; j < coarseModel->nTimeSteps; j++)
    {
      PnlVect G_coarse = pnl_vect_wrap_mat_row(coarseModel->Gincr_drift, j);
      PnlMat G_fine = pnl_mat_wrap_mat_rows(mod->Gincr_drift, j * numberOfStepsL
      pnl_mat_sum_vect(&G_coarse, &G_fine, 'r');
      pnl_vect_div_double(&G_coarse, std::sqrt(double(numberOfStepsLevel1)));
    }
  coarseModel->path();
  payoff_coarse = coarseOption->payoff(coarseModel->pathMatrix);
  return payoff_fine - payoff_coarse;
}

MonteCarloModeReduced::MonteCarloModeReduced(): mod(NULL), opt(NULL), BT(NULL) {

MonteCarloModeReduced::MonteCarloModeReduced(const Param &P)
{
  mod = instantiate_model(P);
  opt = instantiate_option(P);
  IsSize = mod->brownianSize;
  BT = pnl_vect_create(mod->brownianSize);
}
```

```
MonteCarloModeReduced::~MonteCarloModeReduced()
{
  if (BT != NULL) pnl_vect_free(&BT);
  if (mod != NULL) delete mod;
  if (opt != NULL) delete opt;
}

void MonteCarloModeReduced::print(bool verbose) const
{
  std::cout << " Reduced mode" << std::endl;
  if (verbose)
    {
      mod->print();
      opt->print();
    }
  std::cout << "**** Monte Carlo Characteristics ****" << std::endl;
  std::cout << " Number of timesteps " << mod->nTimeSteps << std::endl;
}

void MonteCarloModeReduced::path(PnlRng *rng) const { mod->path(rng); }

void MonteCarloModeReduced::path(PnlRng *rng, const PnlVect *drift) const { mod-

void MonteCarloModeReduced::path() const { mod->path(); }

void MonteCarloModeReduced::path(const PnlVect *drift) const { mod->path(drift);

double MonteCarloModeReduced::payoff() const { return opt->payoff(getSamplePath(

double MonteCarloModeReduced::gaussianWeight(const PnlMat *G, const PnlVect *x,
{
  pnl_mat_sum_vect(BT, G, 'r');
  double scalar = pnl_vect_scalar_prod(BT, x) * mod->sqrt_dt;
  double norm = pnl_vect_scalar_prod(x, x) * mod->maturity;
  if (isplus) return exp(-scalar + norm / 2.);
  else return exp(-scalar - norm / 2);
}

void MonteCarloModeReduced::gaussianGradWeight(PnlVect *res, const PnlMat *G, co
{
  pnl_mat_sum_vect(BT, G, 'r');
```

```
  pnl_vect_div_double(BT, mod->sqrt_nTimeSteps);
  pnl_vect_clone(res, BT);
  pnl_vect_mult_double(res, - 1. / sqrt(mod->maturity) *
                       exp(-2 * (pnl_vect_scalar_prod(BT, x))
                           * sqrt(mod->maturity) - pnl_vect_scalar_prod(x, x) *
}

PnlVect* MonteCarloModeReduced::getGaussianIsVariable(const PnlMat *G) const
{
  PnlVect *g  = pnl_vect_new();
  pnl_mat_sum_vect(g, G, 'r');
  pnl_vect_div_double(g, mod->sqrt_nTimeSteps);
  return g;
}

MonteCarloModeReducedMultiLevel::MonteCarloModeReducedMultiLevel() :
  MonteCarloModeReduced(), coarseModel(NULL), coarseOption(NULL) { }

MonteCarloModeReducedMultiLevel::MonteCarloModeReducedMultiLevel(const Param &P)
  MonteCarloModeReduced(P)
{
  int fine_timesteps;
  Param Pcopy(P);
  P.extract("timestep number level one", numberOfStepsLevel1);
  P.extract("timestep number", fine_timesteps);
  Pcopy.set("timestep number", fine_timesteps / numberOfStepsLevel1);
  coarseModel = instantiate_model(Pcopy);
  coarseOption = instantiate_option(Pcopy);
}

MonteCarloModeReducedMultiLevel::~MonteCarloModeReducedMultiLevel()
{
  if (coarseModel != NULL) delete coarseModel;
  if (coarseOption != NULL) delete coarseOption;
}

/**
 * Compute the difference of the payoffs on the fine and coarse grids
 *
 * @return
 */
```

```
double MonteCarloModeReducedMultiLevel::payoff() const
{
  double payoff_coarse, payoff_fine;
  payoff_fine = opt->payoff(mod->pathMatrix);
  // Agregate steps
  for (int j = 0; j < coarseModel->nTimeSteps; j++)
    {
      PnlVect G_coarse = pnl_vect_wrap_mat_row(coarseModel->Gincr_drift, j);
      PnlMat G_fine = pnl_mat_wrap_mat_rows(mod->Gincr_drift, j * numberOfStepsL
      pnl_mat_sum_vect(&G_coarse, &G_fine, 'r');
      pnl_vect_div_double(&G_coarse, std::sqrt(double(numberOfStepsLevel1)));
    }
  coarseModel->path();
  payoff_coarse = coarseOption->payoff(coarseModel->pathMatrix);
  return payoff_fine - payoff_coarse;
}

MonteCarloJumpModeReduced::MonteCarloJumpModeReduced() : MonteCarloModeReduced()

MonteCarloJumpModeReduced::MonteCarloJumpModeReduced(const Param &P) :
  MonteCarloModeReduced(P)
{
  dynamic_model = dynamic_cast<JumpModel *>(mod);
  if (dynamic_model != NULL)
    {
      NT = pnl_vect_create(dynamic_model->poissonSize);
    }
  else
    {
      std::cout << "Cannot convert BaseModel to JumpModel" << std::endl;
      abort();
    }
}

MonteCarloJumpModeReduced::~MonteCarloJumpModeReduced()
{
  if (NT != NULL) pnl_vect_free(&NT);
}

void MonteCarloJumpModeReduced::pathMuTheta(PnlRng *rng, const PnlVect *drift, c
{
```

```
  return dynamic_model->pathMuTheta(rng, drift, mu);
}

void MonteCarloJumpModeReduced::pathMu(PnlRng *rng, const PnlVect *mu) const
{
  return dynamic_model->pathMu(rng, mu);
}

double MonteCarloJumpModeReduced::poissonWeight(const PnlVect *lambda, const Pnl
{
  pnl_mat_sum_vect(NT, myPoiss, 'r');
  double prod = 1.0;
  for (int i = 0 ; i < lambda->size ; i++)
    {
      double lambda_i = GET(lambda, i);
      double mu_i = GET(mu, i);
      double N_i = GET(NT, i);
      prod *= exp((mu_i - lambda_i) * mod->maturity) * pow((lambda_i / mu_i), N_
    }
  return prod;
}

PnlVect* MonteCarloJumpModeReduced::getPoissonIsVariable(const PnlMat *P) const
{
  PnlVect *n  = pnl_vect_new();
  pnl_mat_sum_vect(n, P, 'r');
  return n;
}
```