Help

```
/*
 * File written by Jérôme Lelong <jerome.lelong@gmail.com>
 * for Premia release 11
 * February 2009
 */

#include <stdlib.h>
#include "pnl/pnl_mathtools.h"
#include "pnl/pnl_random.h"
#include "pnl/pnl_cdf.h"
#include "pnl/pnl_matrix.h"
#include "pnl/pnl_vector.h"

#include "bsnd_stdnd.h"
#include "enums.h"
#include "math/bsnd_math/bsnd_path.h"

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion <
     (2009+2) //The "#else" part of the code will be freely av
    ailable after the (year of creation of this file + 2)
static int CHK_OPT(MC_Jourdain_Lelong)(void *Opt, void *
    Mod)
{
  return NONACTIVE;
}
int CALC(MC_Jourdain_Lelong)(void *Opt, void *Mod, Pricing
    Method *Met)
{
  return AVAILABLE_IN_FULL_PREMIA;
}
#else

/*
 * Call the payoff function stored in a NumFunc_nd
 */
static double payoff_func (const PnlMat *path, NumFunc_nd *
    p)
{
  /* create a wrapper for the final values of St */
  PnlVect St;
```

```
  St.size = path->n; St.owner = 0;
  St.array = &(path->array[(path->m-1)*path->n]); /* path(
    m-1, 0) */
  return p->Compute (p->Par, &St);
}


/**
 * Computes
 * expect_0 = E(payoffs^2 * exp(-theta . g sqrt(T)))
 * expect_1 = E(payoffs^2 * g * exp(-theta . g sqrt(T)))
 * expect_0 = E(payoffs^2 * g' * g * exp(-theta . g sqrt(T)
    ))
 *
 * @param g is an array of PnlVect. g[i] is the vector of
    W_T / sqrt(T).
 * @param theta the drift vector
 * @param payoffs the vector of payoff values
 * @param T the maturity time
 * @param N the number of samples
 * @param expect_0 a double containing E(payoffs^2 * exp(-
    theta . g sqrt(T))) on exit
 * @param expect_1 a vector containing  E(payoffs^2 * g *
    exp(-theta . g sqrt(T))) on exit
 * @param expect_2 a matri containing  E(payoffs^2 * g' *
    g * exp(-theta . g sqrt(T))) on exit
 */
static void expectation_order_n(  PnlVect * const * g, cons
    t PnlVect *theta,
                                  const PnlVect *payoffs,
    double T, int N, double *expect_0,
                                  PnlVect *expect_1, PnlMa
    t *expect_2 )
{
  double tmp;
  int i;
  double payoffs_i;
  *expect_0 = 0.0;
  pnl_vect_set_double (expect_1, 0.);
  pnl_mat_set_double (expect_2, 0.);
```

```
  for (i=0; i<N; i++)
    {
      payoffs_i = pnl_vect_get (payoffs, i);
      tmp = payoffs_i * payoffs_i * exp (-pnl_vect_scalar_
    prod (theta, g[i]) * sqrt (T));
      *expect_0 += tmp;
      pnl_vect_axpby (tmp, g[i], 1., expect_1); /* E1 += tm
    p  * g[i] */
      pnl_mat_dger (tmp, g[i], g[i], expect_2); /* E2 += tm
    p  * g[i]' * g[i] */
    }
  *expect_0 /= N;
  pnl_vect_mult_double (expect_1, sqrt (T) / N);
  pnl_mat_mult_double (expect_2, T / N);
}

/**
 * Find the optimal theta
 *
 * @param g is an array of PnlVect. g[i] is the vector of
    W_T / sqrt(T).
 * @param theta the drift vector
 * @param payoffs the vector of payoff values
 * @param d the size of the model
 * @param T the maturity time
 * @param N the number of samples
 */
static void sample_averaging_newton (PnlVect *theta,  PnlV
    ect * const *g, const PnlVect *payoffs,
                                     int d, int N, double
    T)
{
  double expect_0, norm_gradv;
  PnlVect *expect_1, *grad_v;
  PnlMat *expect_2, *hes_v;
  int l;
  double EPS = 0.00000001*d;
  int k=30;

  expect_1 = pnl_vect_create (d);
  grad_v = pnl_vect_create (d);
```

```
  expect_2 = pnl_mat_create (d, d);
  hes_v = pnl_mat_create (d, d);
  pnl_vect_resize (theta, d);
  pnl_vect_set_double (theta, 0.);

  for (l=0; l<k; l++)
    {
      expectation_order_n( g, theta, payoffs, T, N, &expec
    t_0, expect_1, expect_2 );

      pnl_vect_clone (grad_v, theta);
      pnl_vect_axpby (1. / (-T * expect_0), expect_1, 1.,
    grad_v);

      /* hes_v = I + ( E2 E 0 + E1'E1) / (E0^2 T) */
      pnl_mat_div_double (expect_2, expect_0 * T);
      pnl_mat_set_id (hes_v);
      pnl_mat_plus_mat (hes_v, expect_2);
      pnl_mat_dger ( -1. / (expect_0 * expect_0 * T), expec
    t_1, expect_1, hes_v);

      norm_gradv = pnl_vect_norm_two (grad_v);
      pnl_mat_chol (hes_v);
      pnl_mat_chol_syslin_inplace (hes_v, grad_v);
      pnl_vect_axpby (-1., grad_v, 1., theta); /* theta -=
    grad_v */
      if (norm_gradv < EPS) break;
    }
  pnl_vect_free (&expect_1);
  pnl_vect_free (&grad_v);
  pnl_mat_free (&expect_2);
  pnl_mat_free (&hes_v);
}

/**
 * Monte Carlo with importance Sampling
 * The optimal importance smapling parameter is determined
    using sample
 * averaging techniques rather than stochastic approximatio
    n. Then the Monte
 * Carlo approximation is computed using the same samples
```

```
    as in the sample
 * average approximation step.
 *
 * @param mod a B&S structure
 * @param T the maturity time
 * @param N the number of samples
 * @param gen the index of the random generator to be used
 * @param price a double containig the price on exit
 * @param var a double containig the variance on exit
 */
static void mc_sample_averaging(const PremiaBSnd *mod,
    double T, int N,int gen,
                                NumFunc_nd *p, double *
    price, double *var)
{
  PnlVect *theta, *payoffs;
  PnlVect **g_final;
  PnlMat **G, *path;
  double  tmp, sqrt_T, sqrt_timesteps;
  int i;

  pnl_rand_init (gen, N, mod->d);
  payoffs = pnl_vect_create (N);
  theta = pnl_vect_create (N);
  G = malloc (sizeof (PnlMat*) * N);
  g_final = malloc (sizeof (PnlVect*) * N);
  path = pnl_mat_create (0, 0);
  *price=0.0; *var=0.0;
  sqrt_T = sqrt (T);
  sqrt_timesteps = sqrt (mod->timesteps);
  /*
   * Draw a set of N B&S paths
   * On each path, compute and store the payoff
   */
  for (i=0; i<N; i++)
    {
      G[i] = pnl_mat_create (0, 0);
      pnl_mat_rand_normal(G[i], mod->timesteps, mod->d,
    gen);
      premia_bs_path (path, mod , G[i], T, NULL);
      g_final[i] = pnl_vect_create (0);
```

```
      pnl_mat_sum_vect (g_final[i], G[i], 'r');
      pnl_vect_div_double (g_final[i], sqrt_timesteps);
      pnl_vect_set (payoffs, i, payoff_func (path, p));
    }

  /*
   * computation of the theta optimal
   */
  sample_averaging_newton (theta, g_final, payoffs, mod->d,
     N, T);

  /*
   * Computation of MC with that value using the same sampl
    es
   */
  for (i=0; i<N; i++)
    {
      premia_bs_path (path, mod , G[i], T, theta);
      tmp = payoff_func (path, p) * exp (- pnl_vect_scalar_
    prod (g_final[i], theta) * sqrt_T -
                                         pnl_vect_scalar_
    prod (theta, theta) * T / 2. );
      *price += tmp;
      *var += tmp * tmp;
    }
  *price *= exp(-mod->r * T) / N;
  *var = *var * exp(-2. * mod->r * T) / N - *price * *
    price;

  pnl_vect_free (&theta);
  pnl_vect_free (&payoffs);
  pnl_mat_free (&path);
  for (i=0; i<N; i++)
    {
      pnl_vect_free (&g_final[i]);
      pnl_mat_free (&G[i]);
    }
  free (g_final);
  free (G);
}
```

```
int CALC(MC_Jourdain_Lelong)(void *Opt, void *Mod, Pricing
    Method *Met)
{
  TYPEOPT* ptOpt=(TYPEOPT*)Opt;
  TYPEMOD* ptMod=(TYPEMOD*)Mod;
  double r;
  int i, size;
  PnlVect *spot, *sig, *divid;
  PnlMat *LGamma;
  PremiaBSnd mod;
  double alpha, z_alpha, var, price, inf_price, sup_price,
    error_price;

  size = ptMod->Size.Val.V_PINT;
  divid = pnl_vect_create(size);
  spot = pnl_vect_compact_to_pnl_vect (ptMod->S0.Val.V_PNLV
    ECTCOMPACT);
  sig = pnl_vect_compact_to_pnl_vect (ptMod->Sigma.Val.V_PN
    LVECTCOMPACT);

  for(i=0; i<size; i++)
    pnl_vect_set (divid, i,
                  log(1.+ pnl_vect_compact_get (ptMod->Div
    id.Val.V_PNLVECTCOMPACT, i)/100.));

  r= log(1.+ptMod->R.Val.V_DOUBLE/100.);

  LGamma = pnl_mat_create_from_double (size, size, ptMod->
    Rho.Val.V_DOUBLE);
  for (i=0; i<size; i++) pnl_mat_set (LGamma, i, i, 1.);
  pnl_mat_chol (LGamma);
  mod.spot = spot;
  mod.LGamma = LGamma;
  mod.sigma = sig;
  mod.r = r;
  mod.divid = divid;
  mod.d = ptMod->Size.Val.V_PINT;
  mod.timesteps = 1;

  mc_sample_averaging(&mod,
```

```
                              ptOpt->Maturity.Val.V_DATE-ptMod->T.
    Val.V_DATE,
                              Met->Par[1].Val.V_PINT,

                              Met->Par[0].Val.V_ENUM.value,
                              ptOpt->PayOff.Val.V_NUMFUNC_ND,
                              &price, &var);

  Met->Res[0].Val.V_DOUBLE = price;
  /* Value to construct the confidence interval */
  alpha= (1. - Met->Par[2].Val.V_DOUBLE) / 2.;
  z_alpha= pnl_inv_cdfnor(1.- alpha);
  error_price = sqrt (var /  Met->Par[1].Val.V_PINT);
  inf_price = price - z_alpha * error_price;
  sup_price = price + z_alpha * error_price;

  Met->Res[1].Val.V_DOUBLE = error_price;
  Met->Res[2].Val.V_DOUBLE = inf_price;
  Met->Res[3].Val.V_DOUBLE = sup_price;

  pnl_vect_free(&divid);
  pnl_vect_free (&spot);
  pnl_vect_free (&sig);
  pnl_mat_free (&LGamma);
  return OK;
}

static int CHK_OPT(MC_Jourdain_Lelong)(void *Opt, void *
    Mod)
{
  Option* ptOpt=(Option*)Opt;
  TYPEOPT* opt=(TYPEOPT*)(ptOpt->TypeOpt);

  if ( (strcmp(ptOpt->Name,"CallBasketEuro_nd")==0) ||
       (strcmp( ptOpt->Name,"PutBasketEuro_nd")==0) )
    return OK;
  if ((opt->EuOrAm).Val.V_BOOL==EURO)
    return OK;

  return WRONG;
}
```

```
#endif //PremiaCurrentVersion
static int MET(Init)(PricingMethod *Met,Option *Opt)
{
  if ( Met->init == 0)
    {
      Met->init=1;
      Met->Par[0].Val.V_ENUM.value = 0;
      Met->Par[0].Val.V_ENUM.members = &PremiaEnumMCRNGs;
      Met->Par[1].Val.V_PINT = 10000;
      Met->Par[2].Val.V_DOUBLE= 0.95;
    }
  return OK;
}

PricingMethod MET(MC_Jourdain_Lelong)=
{
  "MC_JourdainLelong",
  {
    {"RandomGenerator",ENUM,{0},ALLOW},
    {"N iterations",PINT,{10000},ALLOW},
    {"Confidence Value",DOUBLE,{100},ALLOW},
    {" ",PREMIA_NULLTYPE,{0},FORBID}},
  CALC(MC_Jourdain_Lelong),
  {{"Price",DOUBLE,{100},FORBID},
   {"Error Price",DOUBLE,{100},FORBID},
   {"Inf Price",DOUBLE,{100},FORBID},
   {"Sup Price",DOUBLE,{100},FORBID} ,
   {" ",PREMIA_NULLTYPE,{0},FORBID}},
  CHK_OPT(MC_Jourdain_Lelong),
  CHK_ok,
  MET(Init)
};
```

References