Help

```
#include <stdlib.h>
#include  "wishart2d_std2d.h"
#include "pnl/pnl_integration.h"
#include "pnl/pnl_random.h"
#include "pnl/pnl_mathtools.h"
#include "pnl/pnl_vector.h"
#include "pnl/pnl_matrix.h"
#include "pnl/pnl_cdf.h"
#include "enums.h"

static PnlMat *a, *b;
static int i_fix, j_fix;

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2015+2) //The "#els
static int CHK_OPT(MC_BaldiPisani_Wishart)(void *Opt, void *Mod)
{
  return NONACTIVE;
}
int CALC(MC_BaldiPisani_Wishart)(void *Opt, void *Mod, PricingMethod *Met)
{
  return AVAILABLE_IN_FULL_PREMIA;
}
#else

// Camilla Pisani January 2015
// The following code computes prices of European call options on the maximum of
// and put on the minimum
// The model considered is Gorieroux Sufana
// where the volatility follows a Wishart process
// dS_l(t)=r S_l(t)dt+S_l(t)(sqrt(X_t) dB(t))_l    0<=l<=d
// dX_t=(alpha a a^T+b X_t+X_t b^T)dt+ sqrt(X_t) dW_t a + a^T dW_t^T sqrt(X_t)
// The Wishart process is simulated according to the procedure q_2 in the paper
// Asset prices are simulated according to the procedure in [AhdidaAlfonsi]

// Computation of the matrix R= square root of the 2x2 matrix x
static void square_root(PnlMat *x, PnlMat *R)
{
  double delta, tau, t, s;
```

```
  delta = pnl_mat_get(x, 0, 0) * pnl_mat_get(x, 1, 1) - pnl_mat_get(x, 0, 1) * p
  s = sqrt(delta);
  tau = pnl_mat_get(x, 0, 0) + pnl_mat_get(x, 1, 1);
  t = sqrt(tau + 2.*s);

  pnl_mat_set(R, 0, 0, (pnl_mat_get(x, 0, 0) + s) / t);
  pnl_mat_set(R, 0, 1, pnl_mat_get(x, 0, 1) / t);
  pnl_mat_set(R, 1, 0, pnl_mat_get(R, 0, 1));
  pnl_mat_set(R, 1, 1, (pnl_mat_get(x, 1, 1) + s) / t);
}


// Simulation according to the generator L_2
// that is simulation of the square of an Ornstein Uhlenbeck process
static void L2(PnlMat *X_new, PnlMat *EXP, PnlMat *C_half, int n, int generator)
{
  PnlMat *Z, *W, *Y, *R;
  int i, j, k;

  Z = pnl_mat_create_from_double(n, 2, 0.0);
  W = pnl_mat_create_from_double(n, 2, 0.0);
  Y = pnl_mat_create_from_double(n, 2, 0.0);
  R = pnl_mat_create_from_double(2, 2, 0.0);
  // computation of the matrix Y s.t. Y^T*Y=x
  // first step: computation square root of the bidimensional matrix X

  square_root(X_new, R);

  for (i = 0; i < 2; i++)
    for (j = 0; j < 2; j++)
      pnl_mat_set(Y, i, j, pnl_mat_get(R, i, j));

  //Construction matrix W
  for (i = 0; i < n; i++)
    for (j = 0; j < 2; j++)
      pnl_mat_set(W, i, j, pnl_rand_normal(generator));

  //Computation matrix Z
  for (i = 0; i < n; i++)
    for (j = 0; j < 2; j++)
      for (k = 0; k < 2; k++)
```

```
        pnl_mat_set(Z, i, j, pnl_mat_get(Z, i, j) + pnl_mat_get(Y, i, k)*pnl_mat


  //Computation matrix X=Z^T Z
  for (i = 0; i < 2; i++)
    for (j = 0; j < 2; j++)
      {
        pnl_mat_set(X_new, i, j, 0.0);
        for (k = 0; k < n; k++)
          pnl_mat_set(X_new, i, j, pnl_mat_get(X_new, i, j) + pnl_mat_get(Z, k,
      }

  pnl_mat_free(&R);
  pnl_mat_free(&Z);
  pnl_mat_free(&W);
  pnl_mat_free(&Y);
}


//Simulation Wishart at time t_i+1 starting from a simulation at time t_i
//following the composition rule p_2(t/2)*p_1(t)*p_2(t/2)
//that is according to the scheme q_2 in the paper
static void Wishart(PnlMat *X_new, PnlMat *EXP, PnlMat *C_half, int n, PnlMat *M
{
  int i, j;

//application p_2(t/2)
  L2(X_new, EXP, C_half, n, generator);

//application p_1(t)
  for (i = 0; i < 2; i++)
    for (j = 0; j < 2; j++)
      {
        pnl_mat_set(X_new, i, j, pnl_mat_get(X_new, i, j) + pnl_mat_get(M1, i, j
      }
//application p_2(t/2)
  L2(X_new, EXP, C_half, n, generator);

}

//Simulation of the asset
```

```
static void assets(PnlVect *S_new, PnlMat *X_new, double r, PnlVect *divid, doub
{
  PnlVect *B, *S_old;
  PnlMat *R, *X_old;
  int l, k;

  B = pnl_vect_create_from_double(2, 0.0);
  R = pnl_mat_create_from_double(2, 2, 0.0);

  S_old = pnl_vect_copy(S_new);
  X_old = pnl_mat_copy(X_new);
  square_root(X_old, R);
  for (l = 0; l < 2; l++)
    pnl_vect_set(B, l, sqrt(h)*pnl_rand_normal(generator));

  for (l = 0; l < 2; l++)
    {
      pnl_vect_set(S_new, l, (r - pnl_vect_get(divid, l) - MGET(X_new, l, l) * 0
      for (k = 0; k < 2; k++)
        {
          pnl_vect_set(S_new, l, pnl_vect_get(S_new, l) + MGET(R, l, k)*pnl_vect
        }
      pnl_vect_set(S_new, l, exp(pnl_vect_get(S_new, l)));
      pnl_vect_set(S_new, l, pnl_vect_get(S_new, l)*pnl_vect_get(S_old, l));
    }

  pnl_mat_free(&R);
  pnl_mat_free(&X_old);
  pnl_vect_free(&B);
  pnl_vect_free(&S_old);
}

//Computation of C_half
static double chalf(double x, void *p)
{
  PnlMat *bx, *btx, *EXP1, *EXP2, *at, *res1, *res2, *res;
  double y;
  int dim;

  dim = 2;
```

```
  EXP1 = pnl_mat_create_from_double(dim, dim, 0.0);
  EXP2 = pnl_mat_create_from_double(dim, dim, 0.0);

  bx = pnl_mat_copy(b);
  pnl_mat_mult_double(bx, x);
  btx = pnl_mat_transpose(bx);
  at = pnl_mat_transpose(a);

  pnl_mat_exp(EXP1, bx);
  pnl_mat_exp(EXP2, btx);

  res1 = pnl_mat_mult_mat(EXP1, at);
  res2 = pnl_mat_mult_mat(res1, a);
  res = pnl_mat_mult_mat(res2, EXP2);

  y = pnl_mat_get(res, i_fix, j_fix);

  pnl_mat_free(&EXP1);
  pnl_mat_free(&EXP2);
  pnl_mat_free(&bx);
  pnl_mat_free(&btx);
  pnl_mat_free(&at);
  pnl_mat_free(&res1);
  pnl_mat_free(&res2);
  pnl_mat_free(&res);

  return y;
}

static double compute_in(double x, double y)
{
  double abserr, results;
  int neval;
  PnlFunc func;

  func.F = chalf;
  func.params = NULL;

  neval = 100;
  pnl_integration_GK(&func, x, y, 0.0001, 1, &results, &abserr, &neval);
```

```
  return results;

}

int MCBaldiPisaniWishart(PnlVect *S0, NumFunc_2 *p, double T, double r, PnlVect
{
  long i, ipath;
  double price_sample, mean_price, var_price;
  int init_mc;
  int simulation_dim;
  double  z_alpha;
  double h;
  double S_T1, S_T2;
  int dim;
  int n, j, l;
  PnlVect  *S_new;
  PnlMat *EXP, *X_0, *X_new, *M1, *C_half, *C;

  dim = 2;
  n = floor(alpha);

  //----------------------Initialisation of variable
  X_0 = pnl_mat_create_from_double(dim, dim, 0.0);
  X_new = pnl_mat_create_from_double(dim, dim, 0.0);
  a = pnl_mat_create_from_double(dim, dim, 0.0);
  b = pnl_mat_create_from_double(dim, dim, 0.0);
  EXP = pnl_mat_create_from_double(dim, dim, 0.0);
  C_half = pnl_mat_create_from_double(dim, dim, 0.0);
  C = pnl_mat_create_from_double(dim, dim, 0.0);
  S_new = pnl_vect_create_from_double(dim, 0.0);
  MLET(X_0, 0, 0) = GET(X_0V, 0);
  MLET(X_0, 0, 1) = GET(X_0V, 1);
  MLET(X_0, 1, 0) = GET(X_0V, 2);
  MLET(X_0, 1, 1) = GET(X_0V, 3);

  MLET(a, 0, 0) = GET(aV, 0);
  MLET(a, 0, 1) = GET(aV, 1);
  MLET(a, 1, 0) = GET(aV, 2);
  MLET(a, 1, 1) = GET(aV, 3);

  MLET(b, 0, 0) = GET(bV, 0);
```

```
MLET(b, 0, 1) = GET(bV, 1);
MLET(b, 1, 0) = GET(bV, 2);
MLET(b, 1, 1) = GET(bV, 3);

//Time Step
h = T / nstep;

//Compute C_half
for (i_fix = 0; i_fix < dim; i_fix++)
  for (j_fix = 0; j_fix < dim; j_fix++)
    {
      pnl_mat_set(C, i_fix, j_fix, compute_in(0, h / 2.));
    }

square_root(C, C_half);

//Exponential Matrix
pnl_mat_mult_double(b, h / 2.);
pnl_mat_exp(EXP, b);

//Computation of Matrix Shift
pnl_mat_mult_double(b, 2. / h);
M1 = pnl_mat_mult_mat(a, pnl_mat_transpose(a));
pnl_mat_mult_scalar(M1, (alpha - n)*h);

/* Value to construct the confidence interval */
alpha = (1. - confidence) / 2.;
z_alpha = pnl_inv_cdfnor(1. - alpha);

/*Initialisation*/
mean_price = 0.0;
var_price = 0.0;

/* Size of the random vector we need in the simulation */
simulation_dim = nb;

/* MC sampling */
init_mc = pnl_rand_init(generator, simulation_dim, nstep);
/* Test after initialization for the generator */
if (init_mc == OK)
  {
```

```
for (ipath = 1; ipath <= nb; ipath++)
  {
    //Simulation of a path of (S_1,S_2)
    //First step:initialization
    for (l = 0; l < 2; l++)
      pnl_vect_set(S_new, l, pnl_vect_get(S0, l));
    for (j = 0; j < 2; j++)
      for (l = 0; l < 2; l++)
        pnl_mat_set(X_new, j, l, pnl_mat_get(X_0, j, l));

    for (i = 0; i < nstep; i++)
      {
        if (pnl_rand_uni(generator) < 0.5)
          {
            //Update the assets
            assets(S_new, X_new, r, divid, h, generator);
            S_T1 = pnl_vect_get(S_new, 0);
            S_T2 = pnl_vect_get(S_new, 1);

            //Update vol

            Wishart(X_new, EXP, C_half, n, M1, generator);

          }
        else
          {
            //Update the vol
            Wishart(X_new, EXP, C_half, n, M1, generator);

            //Update the assets
            assets(S_new, X_new, r, divid, h, generator);
          }
      }

    /*Price*/
    S_T1 = pnl_vect_get(S_new, 0);
    S_T2 = pnl_vect_get(S_new, 1);
    price_sample = (p->Compute)(p->Par, S_T1, S_T2);

    /* Sum */
```

```
          mean_price += price_sample;

          /* Sum of squares */
          var_price += SQR(price_sample);
        }
      /* End of the N iterations */

      /* Price estimator */
      *ptprice = (mean_price / (double)nb);
      *pterror_price = exp(-r * T) * sqrt(var_price / (double)nb - SQR(*ptprice)
      *ptprice = exp(-r * T) * (*ptprice);

      /* Price Confidence Interval */
      *inf_price = *ptprice - z_alpha * (*pterror_price);
      *sup_price = *ptprice + z_alpha * (*pterror_price);
    }

  ///----------------Desallocation memory
  pnl_vect_free(&S_new);
  pnl_mat_free(&X_0);
  pnl_mat_free(&X_new);
  pnl_mat_free(&a);
  pnl_mat_free(&b);
  pnl_mat_free(&EXP);
  pnl_mat_free(&M1);
  pnl_mat_free(&C_half);
  pnl_mat_free(&C);

  return init_mc;
}

int CALC(MC_BaldiPisani_Wishart)(void *Opt, void *Mod, PricingMethod *Met)
{
  TYPEOPT *ptOpt = (TYPEOPT *)Opt;
  TYPEMOD *ptMod = (TYPEMOD *)Mod;
  double r;

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

  return MCBaldiPisaniWishart(ptMod->S0.Val.V_PNLVECT,
                              ptOpt->PayOff.Val.V_NUMFUNC_2,
```

```
                                ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE,
                                r,  ptMod->Divid.Val.V_PNLVECT, ptMod->alpha.Val.V
                                ptMod->Sigma0.Val.V_PNLVECT,
                                ptMod->Q.Val.V_PNLVECT,
                                Met->Par[0].Val.V_LONG,
                                Met->Par[1].Val.V_INT,
                                Met->Par[2].Val.V_ENUM.value,
                                Met->Par[3].Val.V_PDOUBLE,
                                &(Met->Res[0].Val.V_DOUBLE),
                                &(Met->Res[1].Val.V_DOUBLE),
                                &(Met->Res[2].Val.V_DOUBLE),
                                &(Met->Res[3].Val.V_DOUBLE));

}



static int CHK_OPT(MC_BaldiPisani_Wishart)(void *Opt, void *Mod)
{
  if ((strcmp(((Option *)Opt)->Name, "CallMaximumEuro") == 0) || (strcmp(((Optio
    return OK;

  return  WRONG;
}
#endif //PremiaCurrentVersion

static int MET(Init)(PricingMethod *Met, Option *Opt)
{
  //int type_generator;
  if (Met->init == 0)
    {
      Met->init = 1;

      Met->Par[0].Val.V_LONG = 100000;
      Met->Par[1].Val.V_INT = 10;
      Met->Par[2].Val.V_ENUM.value = 0;
      Met->Par[2].Val.V_ENUM.members = &PremiaEnumMCRNGs;
      Met->Par[3].Val.V_DOUBLE = 0.95;
    }
```

```
  return OK;
}


PricingMethod MET(MC_BaldiPisani_Wishart) =
{
  "MC_BaldiPisani_Wishart",
  { {"N iterations", LONG, {100}, ALLOW},
    {"TimeStepNumber", LONG, {100}, ALLOW},
    {"RandomGenerator", ENUM, {100}, ALLOW},
    {"Confidence Value", DOUBLE, {100}, ALLOW},
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
  },
  CALC(MC_BaldiPisani_Wishart),
  { {"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_BaldiPisani_Wishart),
  CHK_mc,
  MET(Init)
};
```