Help

```
#include "
href../../../mod/bsnd/bsnd_stdnd/bsnd_stdnd_h_src.pdfbsnd_stdnd.h"
#include "pnl/pnl_cdf.h"
#include "pnl/pnl_matrix.h"

#include <
href../../../common/math/cdo/cdo_math_h_src.pdfmath.h>
#include <float.h>

#if defined(PremiaCurrentVersion) && PremiaCurrentVersion < (2008+2) //The "#els
static int CHK_OPT(AP_CarmonaDurrleman)(void *Opt, void *Mod)
{
  return NONACTIVE;
}
int CALC(AP_CarmonaDurrleman)(void *Opt, void *Mod, PricingMethod *Met)
{
  return AVAILABLE_IN_FULL_PREMIA;
}
#else

// Returns a*(ab)/abs(ab) :
static double sgne(double a, double b)
{
  return (a * b >= 0) ? a : -a;
}

// Shifts a and b :
static void perm2(double *a, double *b)
{
  double tmp = *a;
  *a = *b;
  *b = tmp;
}

// Puts b into a, c into b, and d into c :
static void chang3(double *a, double *b, double *c, const double d)
{
  *a = *b;
  *b = *c;
```

```
  *c = d;
}

// Brackets a minimum of function f :
static void minenc(double *ax, double *bx, double *cx, double f(double))
{
  // Given initial bracketing, magnifies the interval so that actual bracketing
  const double PHI = 1.618034; // Default maginifying constant
  const double RLIMIT = 200.0; // Limit of parabolic interpolation
  const double EPS = 1.0e-20; // Precision
  double ulim, u, r, q, fa, fb, fc, fu;

  // Searches minimum in downhill direction defined by ax and bx.
  // Stops when starting going back uphill.
  fa = f(*ax);
  fb = f(*bx);
  if (fb > fa)
    {
      perm2(ax, bx);
      perm2(&fb, &fa); // Downhill direction defined to be from a to b.
    }
  *cx = *bx + PHI * (*bx - *ax); // Magnifying interval : going further downhill
  fc = f(*cx);

  while (fb > fc) // Third point not high enough : still going downhill
    {
      // Tries parabolic interpolation
      r = (*bx - *ax) * (fb - fc);
      q = (*bx - *cx) * (fb - fa);
      // Optimum of the interpolated parabol located at u :
      u = (*bx) - ((*bx - *cx) * q - (*bx - *ax) * r) / (2 * sgne(MAX(fabs(q - r
      // Limit parabolic interpolation
      ulim = *bx + RLIMIT * (*cx - *bx);

      if ((*bx - u) * (u - *cx) > 0) // u is between bx and cx
        {
          fu = f(u);
          if (fu < fc) // Minimum between bx and cx
            {
              *ax = *bx;
              *bx = u; // Bracketing triplet is (bx,u,cx)
```

```
            return;
          }
        else if (fu > fb) // Minimum between ax and u
          {
            *cx = u; // Bracketing triplet is (ax,bx,u)
            return;
          }
        u = *cx + PHI * (*cx - *bx); // Parabolic interpolation was useless
        fu = f(u);
      }

    else if ((*cx - u) * (u - ulim) > 0) // u is between cx and ulimit
      {
        fu = f(u);
        if (fu < fc)
          {
            chang3(bx, cx, &u, u + PHI * (u - *cx)); // Further downhill AND d
            chang3(&fb, &fc, &fu, f(u));
          }
      }

    else if ((u - ulim) * (ulim - *cx) >= 0) // Limits u to its maximum value
      {
        u = ulim;
        fu = f(u);
      }

    else
      {
        u = *cx + PHI * (*cx - *bx); // Default magnification
        fu = f(u);
      }

    chang3(ax, bx, cx, u); // Continues further on downhill
    chang3(&fa, &fb, &fc, fu);
  }
}

// Finds a minimum of one-dimensional function f, bracketed by ax, bx, cx, with
static double min1dim(double ax, double bx, double cx, double f(double), double
{
```

```
int i;
const int ITMAX = 1000; // Maximum nuber of iterations allowed
const double PHI = 0.3819660; // Golden ratio : default step
const double EPS = DBL_EPSILON; // Machine precision

double a, b, d = 0.0, etemp, fu, fv, fw, fx;
double p, q, r, tol1, tol2, u, v, w, x, xm;
// x : where minimum value was found so far
// w : where second least value was found
// v : previous value of w
// u : new trial point
double e = 0.0;

a = (ax < cx) ? ax : cx;
b = (ax > cx) ? ax : cx; // Making a < b
x = w = v = bx;
fx = fw = fv = f(x);

for (i = 0; i < ITMAX; i++)
  {
    xm = 0.5 * (a + b); // [a,b] is the bracketing interval (refined at each i
    tol2 = 2.0 * (tol1 = tol * fabs(x) + EPS);
    if (fabs(x - xm) <= (tol2 - 0.5 * (b - a))) // Done : tolerance attained
      {
        *xmin = x;
        return fx;
      }

    if (fabs(e) > tol1) // Parabolic interpolation using x,w,v
      {
        r = (x - w) * (fx - fv);
        q = (x - v) * (fx - fw);
        p = (x - v) * q - (x - w) * r;
        q = 2.0 * (q - r);
        if (q > 0) p = -p;
        q = fabs(q);
        etemp = e;
        e = d;

        if (fabs(p) >= fabs(0.5 * q * etemp) || p <= q * (a - x) || p >= q * (
          // Parabolic interpolation rejected : default step
```

```
      d = PHI * (e = ((x > xm) ? a - x : b - x));
    else
      {
        d = p / q; // Parabolic step
        u = x + d;
        if (u - a < tol2 || b - u < tol2)
          d = sgne(tol1, xm - x);
      }

  }

else d = PHI * (e = ((x >= xm) ? a - x : b - x)); // Default step

u = (fabs(d) >= tol1) ? x + d : x + sgne(tol1, d);
fu = f(u); // Only function evaluation in the loop

// Redefining bracketing triplet in each case
if (fu <= fx)
  {
    if (u >= x) a = x;
    else b = x;
    chang3(&v, &w, &x, u);
    chang3(&fv, &fw, &fx, fu);
  }

else
  {
    if (u < x) a = u;
    else b = u;

    if (fu <= fw || w == x)
      {
        v = w;
        w = u;
        fv = fw;
        fw = fu;
      }
    else if (fu <= fv || v == x || v == w)
      {
        v = u;
        fv = fu;
```

```
            }

        }

    }
  perror("Too many iterations in min1dim\ n");
  *xmin = x;
  return fx;
}

// Global variables used for communication between "virtual" one-dimensional fun
// derived from function f in min1dir and routine min1dir

static int _n;
static double (*func)(PnlVect *);
static PnlVect *_p;
static PnlVect *_dir;

// One-dimensional virtual function derived from function func
// (which happens to be equal to function f in min1dir)
// in direction _dir

static double f1dim(double x)
{
  int j;
  double val;
  PnlVect *xt = pnl_vect_create(_n);
  for (j = 0; j < _n; j++)
    {
      pnl_vect_set(xt, j, pnl_vect_get(_p,  j) + x * pnl_vect_get(_dir,  j));
    }
  val = func(xt);
  pnl_vect_free(&xt);
  return val;
}

// Finds a minimum of a multidimensional function f in direction dir.
// Minimum is stored in min, its location in p :
static void min1dir(int dim, PnlVect *p, PnlVect *dir, double *min, double f(Pnl
{
  int j;
```

```
  const double TOL = 1.0e-10;
  double xx, xmin, bx, ax;

  // Initialise les variables globales
  _n = dim;
  _p = p;
  _dir = dir;
  func = f;

  // [0,1] is the initial bracketing guess
  ax = 0.0;
  xx = 1.0;
  minenc(&ax, &xx, &bx, f1dim); // Computes an acutal bracketing triplet
  *min = min1dim(ax, xx, bx, f1dim, TOL, &xmin); // Computes the minimum of func
  // (which is the minimum of function f in direction dir)

  for (j = 0; j < dim; j++)
    {
      double d = pnl_vect_get(dir , j);
      pnl_vect_set(dir,  j,  d * xmin);
      pnl_vect_set(p, j, pnl_vect_get(p,  j) + pnl_vect_get(dir,  j));    // Set
    }

}

// Conjugate gradient optimization of function f, given its gradient gradf.
// Minimum is stored in min, its location in p. Tolerance is asked :
static void optigc(int dim, PnlVect *p, double tol, double *min,
                   double f(PnlVect *),
                   void gradf(PnlVect *, PnlVect *))
{
  int i, j;
  /* Scalars used to define directions */
  double        gg, gam, fp, dgg;
  PnlVect   *g     = pnl_vect_create(dim); /* Auxiliary direction :
                                                gradient at the minimum */
  PnlVect   *h     = pnl_vect_create(dim); /* Conjugate direction along
                                                which to minimize */
  PnlVect   *grad  = pnl_vect_create(dim);  /* Gradient */
  const int     ITMAX = 20000;
  const double  EPS   = 1.0e-18;
```

```
fp = f(p);
gradf(p, grad);

pnl_vect_clone(h, grad);
pnl_vect_clone(g, grad);
pnl_vect_mult_double(h , -1.0);
pnl_vect_mult_double(g , -1.0);
pnl_vect_mult_double(grad , -1.0);

for (i = 0; i < ITMAX; i++)
  {
    min1dir(dim, p, h, min, f); // Minimizing along direction h
    if (2.0 * fabs((*min) - fp) <= tol * (fabs((*min)) + fabs(fp) + EPS)) // D
      {
        pnl_vect_free(&g);
        pnl_vect_free(&h);
        pnl_vect_free(&grad);
        return;
      }
    fp = (*min);
    gradf(p, grad); // Computes gradient at point p, location of minimum
    dgg = gg = 0.0;

    /* Computes coefficients applied to new direction for h */
    gg = pnl_vect_scalar_prod(g, g);  /* Denominator */
    dgg = pnl_vect_scalar_prod(grad, grad) + pnl_vect_scalar_prod(g, grad);

    if (gg == 0.0) // Gradient equals zero : done
      {
        pnl_vect_free(&g);
        pnl_vect_free(&h);
        pnl_vect_free(&grad);
        return;
      }
    gam = dgg / gg;

    for (j = 0; j < dim; j++) // Defining directions for next iteration
      {
        pnl_vect_set(g, j, - pnl_vect_get(grad,  j));
        pnl_vect_set(h, j, pnl_vect_get(g,  j) + gam * pnl_vect_get(h,  j));
```

```
        }
    }
  perror("Too many iterations in optigc\ n");
}

// Global variables used for communication between Cost and Gradcost (Cout) func
// and low(up)linearprice routine, in which the functions are used

static int Dim;
static PnlVect *Eps;
static PnlVect *X;
static PnlMat *Rac_C;
static double Echeance;
static PnlVect *Sigma;

// Auxiliary cost function to minimize used in lowlinearprice :
static double Cost(PnlVect *ksi)
{
  int i, j;
  double p = 0, arg = 0;

  double normv = 0;
  for (i = 0; i < Dim + 1; i++)
    {
      double ksi_i = pnl_vect_get(ksi,  i);
      normv += ksi_i * ksi_i;
    }
  normv = sqrt(normv);

  for (i = 0; i < Dim + 1; i++)
    {
      double tmp = 0;
      for (j = 0; j < Dim + 1; j++)
        {
          tmp += pnl_mat_get(Rac_C,  i,  j) * pnl_vect_get(ksi,  j);
        }
      arg = pnl_vect_get(ksi,  Dim + 1) + pnl_vect_get(Sigma,  i) * tmp * sqrt(E
      p += pnl_vect_get(Eps,  i) * pnl_vect_get(X,  i) * cdf_nor(arg);
    }

  return (-1.0 * p); // The function is to be maximized
```

```
}

// Auxiliary gradient of function Cost, used in routine lowlinearprice :
static void Gradcost(PnlVect *ksi, PnlVect *g)
{
  int i, j, k;
  double normv = 0, s;
  for (i = 0; i < Dim + 1; i++)
    {
      double ksi_i = pnl_vect_get(ksi,  i);
      normv += ksi_i * ksi_i;
    }
  normv = sqrt(normv);
  pnl_vect_set(g,  Dim + 1,  0);

  for (j = 0; j < Dim + 1; j++)
    {
      pnl_vect_set(g,  j,  0.);
      for (i = 0; i < Dim + 1; i++)
        {
          double tmp = 0;
          for (k = 0; k < Dim + 1; k++)
            {
              tmp += pnl_mat_get(Rac_C,  i,  k) * pnl_vect_get(ksi,  k);
            }
          s = pnl_normal_density(pnl_vect_get(ksi,  Dim + 1) + pnl_vect_get(Sigm
                                 tmp * sqrt(Echeance) / normv);
          s *= pnl_vect_get(Eps,  i) * pnl_vect_get(X,  i);
          if (j == Dim)
            pnl_vect_set(g, Dim + 1, pnl_vect_get(g,  Dim + 1) + s);
          s *= pnl_vect_get(Sigma,  i) * sqrt(Echeance) / normv;
          s *= pnl_mat_get(Rac_C,  i,  j) - pnl_vect_get(ksi, j) * tmp / (normv
          pnl_vect_set(g, j, pnl_vect_get(g,  j) + s);
        }
      pnl_vect_set(g , j , -pnl_vect_get(g,  j));
    }
  pnl_vect_set(g , Dim + 1 , -pnl_vect_get(g,  Dim + 1));
}

// Computes the price and the deltas of a claim using the lower bound of the
// price for an option
```

```
// that is paying a linear combination of assets :
static void lowlinearprice(int _dim, PnlVect *_eps, PnlVect *_x, PnlMat *_rac_C,
{
  int i, j;
  double arg;
  double normv = 0;
  double tol = 1e-15;
  PnlVect *xopt = pnl_vect_create_from_double(_dim + 2, 1. / sqrt(_dim + 1.));
  // Starting point for optimization : normalized vector
  pnl_vect_set(xopt,  _dim + 1,  0.0);

  // Initializing global variables to parameters of the problem
  Dim = _dim;
  Echeance = _echeance;
  Sigma = _sigma;
  Rac_C = _rac_C;
  Eps = _eps;
  X = _x;

  optigc(Dim + 2, xopt, tol, prix, Cost, Gradcost);

  *prix = -1.0 * (*prix); // Price is the maximum of function

  for (i = 0; i < Dim + 1; i++)
    {
      double xopt_i = pnl_vect_get(xopt,  i);
      normv += xopt_i * xopt_i;
    }
  normv = sqrt(normv);

  for (i = 0; i < Dim; i++)
    {
      double tmp = 0;
      for (j = 0; j < Dim + 1; j++)
        {
          tmp += pnl_mat_get(_rac_C,  i + 1,  j) * pnl_vect_get(xopt,  j);
        }
      arg = pnl_vect_get(xopt,  Dim + 1) + pnl_vect_get(Sigma,  i + 1) * tmp * s
      pnl_vect_set(deltas, i, pnl_vect_get(Eps,  i + 1) * cdf_nor(arg)); // Comp
    }
  pnl_vect_free(&xopt);
```

```
}


// Returning the price and the deltas  of a basket option using its lower bound
static void lower_basket(int put_or_call, int dim, PnlVect *vol, PnlVect *poids,
{
  int i, j;
  // Initializing parameters
  PnlVect *sigma = pnl_vect_create(dim + 1);
  PnlVect *x = pnl_vect_create(dim + 1);
  PnlVect *eps = pnl_vect_create(dim + 1);
  PnlMat *rac_C = pnl_mat_create(dim + 1, dim + 1);

  pnl_vect_set(sigma,  0,  0);
  for (i = 1; i < dim + 1; i++)
    {
      pnl_vect_set(sigma, i, pnl_vect_get(vol,  i - 1));
    }

  pnl_vect_set(x, 0, strike * exp(-tx_int * echeance));
  for (i = 1; i < dim + 1; i++)
    {
      pnl_vect_set(x, i, fabs(pnl_vect_get(poids,  i - 1)) *
                   pnl_vect_get(val_init,  i - 1)*
                   exp(-pnl_vect_get(div,  i - 1)*echeance));
    }

  pnl_vect_set(eps,  0,  -1);
  for (i = 1; i < dim + 1; i++)
    {
      if (pnl_vect_get(poids,  i - 1) < 0) pnl_vect_set(eps,  i,  -1);
      else pnl_vect_set(eps,  i,  1);
    }
  if (put_or_call == 1)
    {
      pnl_vect_mult_double(eps, -1.0);
    }

  if (cor != 1)
    {
      PnlMat *C = pnl_mat_create(dim, dim);
```

```
    //    double *C=new double[dim*dim]; // Correlation matrix
    for (i = 0; i < dim; i++)
      {
        for (j = 0; j < dim; j++)
          {
            if (i == j) pnl_mat_set(C,  i,  j,  1);
            else pnl_mat_set(C,  i,  j,  cor);
          }
      }
    pnl_mat_chol(C);

    for (i = 0; i < dim + 1; i++)
      {
        pnl_mat_set(rac_C,  i,  0,  0);
        pnl_mat_set(rac_C,  0,  i, 0);
      }
    for (i = 1; i < dim + 1; i++)
      {
        for (j = 1; j <= i; j++)
          {
            pnl_mat_set(rac_C, i, j, pnl_mat_get(C,  i - 1,  j - 1));
          }
        for (j = i + 1; j < dim + 1; j++)
          {
            pnl_mat_set(rac_C,  i,  j ,  0);
          }
      }

    /* Correlation was useful only to compute a square root of it */
    pnl_mat_free(&C);

  }
else
  {
    for (i = 0; i < dim + 1; i++)
      {
        pnl_mat_set(rac_C,  i,  0,  0);
        pnl_mat_set(rac_C, i, 1, 1);
        for (j = 2; j < dim + 1; j++)
          {
            pnl_mat_set(rac_C,  i,  j,  0);
```

```
          }
        }
      pnl_mat_set(rac_C,  0,  1,  0);
    }

  lowlinearprice(dim, eps, x, rac_C, sigma, echeance, prix, deltas); // Uses the

  /* In deltas are stored the derivatives along x[i], which differ from those
     along val_init[i] */
  for (i = 0; i < dim; i++)
    {
      double d = pnl_vect_get(deltas,  i);
      pnl_vect_set(deltas, i, d * pnl_vect_get(x,  i + 1) / pnl_vect_get(val_ini
    }

  pnl_vect_free(&eps);
  pnl_vect_free(&x);
  pnl_vect_free(&sigma);
  pnl_mat_free(&rac_C);

}

/*see the documentation for the parameters meaning*/

static int ap_carmonadurrleman(PnlVect *BS_Spot,
                               NumFunc_nd *p,
                               double OP_Maturity,
                               double BS_Interest_Rate,
                               PnlVect *BS_Dividend_Rate,
                               PnlVect *BS_Volatility,
                               double rho,
                               double *LowerPrice,
                               PnlVect *LowerDelta)
{
  int BS_Dimension = BS_Spot->size;
  int put_or_call;
  PnlVect *Weights =  pnl_vect_create_from_double(BS_Dimension, 1. / BS_Dimensio
  double Strike = p->Par[0].Val.V_DOUBLE;

  *LowerPrice = 0.;
```

```
  if ((p->Compute) == &CallBasket_nd)
    put_or_call = 0;
  else
    put_or_call = 1;

  lower_basket(put_or_call, BS_Dimension, BS_Volatility, Weights, BS_Spot, BS_Di


  /*upper_basket(put_or_call,BS_Dimension,BS_Volatility->array,Weights,BS_Spot->

  pnl_vect_free(&Weights);

  return OK;
}

int CALC(AP_CarmonaDurrleman)(void *Opt, void *Mod, PricingMethod *Met)
{
  TYPEOPT *ptOpt = (TYPEOPT *)Opt;
  TYPEMOD *ptMod = (TYPEMOD *)Mod;
  double r;
  int i, res;
  PnlVect *divid = pnl_vect_create(ptMod->Size.Val.V_PINT);
  PnlVect *spot, *sig;

  spot = ptMod->S0.Val.V_PNLVECT;
  sig = ptMod->Sigma.Val.V_PNLVECT;

  for (i = 0; i < ptMod->Size.Val.V_PINT; i++)
    pnl_vect_set(divid, i,
                 log(1. + GET(ptMod->Divid.Val.V_PNLVECT, i) / 100.));

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

  res = ap_carmonadurrleman(spot,
                            ptOpt->PayOff.Val.V_NUMFUNC_ND,
                            ptOpt->Maturity.Val.V_DATE - ptMod->T.Val.V_DATE,
                            r, divid, sig,
                            ptMod->Rho.Val.V_DOUBLE,
                            &(Met->Res[0].Val.V_DOUBLE), Met->Res[1].Val.V_PNLVE
  pnl_vect_free(&divid);
```

```
  return res;
}


static int CHK_OPT(AP_CarmonaDurrleman)(void *Opt, void *Mod)
{
  Option *ptOpt = (Option *)Opt;
  if ((strcmp(ptOpt->Name, "CallBasketEuro_nd") == 0) ||
      (strcmp(ptOpt->Name, "PutBasketEuro_nd") == 0))
    return OK;

  return WRONG;
}

#endif //PremiaCurrentVersion
static int MET(Init)(PricingMethod *Met, Option *Opt)
{
  TYPEOPT *opt = (TYPEOPT *)(Opt->TypeOpt);

  if (Met->init == 0)
    {
      Met->init = 1;
      Met->Res[1].Val.V_PNLVECT = NULL;
    }
  /* some initialisation */
  if (Met->Res[1].Val.V_PNLVECT == NULL)
    Met->Res[1].Val.V_PNLVECT = pnl_vect_create(opt->Size.Val.V_PINT);
  else
    pnl_vect_resize(Met->Res[1].Val.V_PNLVECT, opt->Size.Val.V_PINT);

  return OK;

}

PricingMethod MET(AP_CarmonaDurrleman) =
{
  "AP_CarmonaDurrleman_nd",
  {{" ", PREMIA_NULLTYPE, {0}, FORBID}},
  CALC(AP_CarmonaDurrleman),
  { {"Lower Price", DOUBLE, {100}, FORBID}, {"Lower Delta", PNLVECT, {1}, FORBID
    {" ", PREMIA_NULLTYPE, {0}, FORBID}
```

```
  },
  CHK_OPT(AP_CarmonaDurrleman),
  CHK_ok,
  MET(Init)
};
```